TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: Elektrotechnika a informatika
Studijní obor: Informační technologie

JavaScriptový XML editor

JavaScript XML Editor

Diplomová práce

Autor: **Bc. Jindřich Růžička**
Vedoucí práce: Ing. Pavel Tyl

V Liberci 21. 5. 2010

Zadání práce

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 — školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum

Podpis

Poděkování

Mé poděkováni patří všem, kteří se jakýmkoliv způsobem podíleli na tvorbě mé diplomové práce. Zejména pak tímto děkuji mému vedoucímu diplomové práce Ing. Pavlu Tylovi za konzultace, pomoc při vypracování, odborné rady a zkušenosti, které přispěly ke konečné verzi práce.

Abstrakt

Úkolem této práce je napsat jednoduchý *XML* editor v jazyce JavaScript. Editor by měl běžet na straně klienta a obejít se tak bez serveru. Cílem projektu je vyvinout editor tak, aby jej mohl využívat i uživatel bez podrobné znalosti jazyka *XML*.

Jedním z úkolů je prozkoumání jazyka JavaScript jako nástroje pro vývoj rozsáhlejších projektů. Důležitá je technologie objektového programování a volba správných postupů, pomocí kterých se dá tato technika v JavaScriptu realizovat.

V práci je využita volně dostupná JavasScriptová knihovna *jQuery*, která se v poslední době stala populární na poli programování webových aplikací v JavaScriptu. Dále je v práci použita technologie *XSLT*, která umožňuje nasazení WYSIWYG režimu. Ten se hodí zejména v případě kompaktních webových editorů. Aplikace je navržená tak, aby byla dobře rozšiřitelná formou *jQuery pluginů*. Jednotlivé její části lze snadno využít i v jiných aplikacích. To může usnadnit práci i ostatním vývojářům webových editorů.

Klíčová slova

wysiwyg editor, jquery, javascript, www, xhtml, xml, xslt

Abstract

The purpose of this work is to develop simple *XML* editor in JavaScript. Another objective is to make an editor that everybody can use it even without a deep knowledge of *XML* language.

One of the goals is to inspect the JavaScript language as a tool for developing large projects. Important is the technology of objective oriented programming and choosing the right method to achieve this technique in JavaScript.

I use the free JavaScript library *jQuery* in this work. It has been popular in the domain of web application development in last days. I also use the *XSLT* technology for the *WYSIWYG* mode realization which is popular mostly in compact web editors. The application is designed to be easilly expandable with custom jQuery plugins. The single parts of editor could be also used in another applications which could could make the work easier for another web editor developers.

Keywords

wysiwyg editor, jquery, javascript, www, xhtml, xml, xslt

Obsah

Seznam obrázků

Seznam symbolů, zkratek a termínů

API Application Programming Interface – programové rozhraní aplikace, sbírka procedur, funkcí, objektů, které může programátor využívat v případě, že používá danou knihovnu.

CSS Cascading Style Sheets – jazyk pro popis způsobu zobrazení stránek napsaných v jazycích HTML, XHTML nebo XML.

DOM Document Object Model – objektově orientovaná reprezentace XML nebo HTML dokumentu.

FILO First In Last Out – druh zásobníku, ve kterém je prvně přidaný prvek přístupný jako poslední.

GPL GNU General Public License – licence pro svobodný software.

GUI GUI Graphical User Interface – grafická forma uživatelského rozhraní.

HTTP HyperText Transfer Protocol – internetový protokol určený původně pro výměnu hypertextových dokumentů ve formátu HTML.

JS JavaScript – vysokoúrovňový skriptovací programovací jazyk, používaný převážně při vývoji webových aplikací.

JSON JavaScript Object Notification – jazyk pro popis JavaScriptových objektů.

PDF Portable Document Format – souborový formát vyvinutý firmou Adobe pro ukládání dokumentů nezávisle na softwaru i hardwaru, na kterém byly pořízeny.

RE Rich Edit – JavaScriptová komponenta pro Wysiwyg editory, výsledek mého magisterského projektu.

SAX Simple API for XML – sekvenční programové rozhraní pro práci s XML, alternativa k DOM.

UML Unified Modeling Language – grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů.

URL Uniform Resource Locator – slouží k přesné specifikaci umístění zdrojů informací na Internetu.

VCL Viual Component Library – knihovna komponent dostupná ve vývojovém prostředí Delphi.

W3C World Wide Web Consortium – mezinárodní konsorcium pro vývoj webových standardů.

WWW World Wide Web – označení pro aplikace internetového protokolu HTTP.

WYSIWYG What You See Is What You Get – označuje způsob editace dokumentů v počítači, při kterém je verze zobrazená na obrazovce vzhledově totožná s výslednou verzí dokumentu.

XHTML eXtensible HyperText Markup Language – značkovací jazyk pro tvorbu hypertextových dokumentů v prostředí WWW vyvinutý W3C.

XML eXtensible Markup Language – obecný rozšiřitelný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C.

XSLT eXtensible Stylesheet Language Transformation – jazyk založený na XML, definuje, jak se data formátu XML převedou do libovolné jiné datové struktury (nejčastěji HTML, XML) na základě stylového předpisu.

1 Úvod

V rámci své bakalářské práce jsem vytvořil jednoduchý *WYSIWYG* editor pracující s dokumenty napsanými v jazyce *Docbook* během jejich sazby na webové stránce. Důležité body v zadání byly: použít jazyk *JavaScript* (dále *JS*) a *XSLT* (proces převodu *XML* dokumentu např. na *HTML* dokument). Jazyk *JS* byl zvolen proto, aby editor mohl pracovat bez *HTTP* serveru. Během této práce jsem také ukázal možnosti generování různých výstupních formátů ze zdrojového *Docbook* dokumentu.

Po dokončení práce jsem se rozhodl dále pracovat na tomto projektu a vyvíjet aplikaci jako editor obecné *XML* sémantiky. Už během bakalářské práce jsem se nejprve snažil pojmout editor jako obecný *XML* editor, kterému pak jednoduše přidám podporu *Docbooku*. Velmi brzy se to ukázalo jako nelehký úkol, především kvůli *WYSIWYG* režimu, jež tvořil důležitou část zadání. Nakonec jsem aplikaci napsal jako editor pouze pro jazyk *Docbook* a splnil tak sice zadání, nicméně přestože je editor funkční, jeho implementace nebyla nejšťastnější.

Začal jsem přemýšlet, proč je problém napsat *WYSIWYG* editor v jazyce *JS* pro „ne-*HTML*“ značkovací jazyky, vždyť dnešní web je plný jazyka *XML* a *XHTML*. Jeden ze zásadních problému je absence „rozumného“ *HTML* prvku, který by umožňoval editaci formátovaného textu — základ *WYSIWYG* editoru. Důležité je, aby různé části textu byly formátovány podle různých pravidel. Protože vyvíjím *XML* editor, všechny operace s dokumentem jsou prováděny na úrovni zdrojového *XML* kódu, nikoliv *HTML*, které představuje jakousi vizualizaci *XML* pro umožnění *WYSIWYG* režimu. K těmto účelům se *API* navržené pro *HTML* editory nehodí, což je i důvodem proč má původní myšlenka napsat bakalářskou práci jako obecný *XML* editor neuspěla.

Na začátků svého magisterského studia jsem začal pracovat na vlastním *API* pro *WYSIWYG* edtiory, které poskytuje lepší možnosti než *API* prohlížeče a řeší některé nekompatibility. Tuto knihovnu jsem napsal jako *plugin* pro framework *jQuery*, který dnes patří mezi nejpopulárnější frameworky pro jazyk *JS*. Výsledný *plugin* umožňuje například vývojářům reagovat pomocí vlastních skriptů na určité operace s dokumentem, potlačovat je či jakkoliv přizpůsobovat. Dále umožňuje pohodlné nastavování resp. zjišťování pozice kursoru (neboli *caret*) v dokumentu. *Plugin* je užitečný jak pro tento projekt, tak pro libovolný webový *WYSIWYG* editor. Komponentu kterou realizuje tento *plugin* jsem nazval `RichEdit` (dále *RE*). Práci jsem obhájil jako magisterský ročníkový projekt.

Tato práce je pokračováním předchozí, jednotlivé části budou reprezentovány dalšími pluginy pro *jQuery* tak, aby ve výsledku tvořily webový *XML* editor.

2 Objektově orientované programování a JavaScript

2.1 Vznik objektového programování

Šedesátá léta dvacátého století přinesla programovací jazyky třetí generace a jednalo se o jeden z největších zlomů v historii programování. Jsou to strojově nezávislé programovací jazyky, podporující metody *strukturovaného programování*. Ty jsou přenositelné a rozčleněné do menších autonomních modulů, které obsahují rozhraní a tělo. Příkladem modulu je procedura (resp. funkce), jejímž rozhraním jsou její parametry a tělem je její implementace. Prvními takovými jazyky byly *Fortran* (IBM 1957), *Algol* a *Cobol*. Jazyk *PL/1* z poloviny 60. let byl pokusem o co nejuniverzálnější programovací jazyk, nicméně později vznikl jiný univerzální avšak jednodušší jazyk — *Pascal*, používaný dodnes zejména k výuce. Asi nejvýznamnějším jazykem této generace je jazyk *C*, který položil standard pro syntaktická pravidla mnoha budoucích jazyků, včetně jazyku *JavaScript*. Na mikropočítačích se prosadil snadno zvládnutelný *Basic* a také se objevily databázové programovací jazyky, např. jazyk *SQL*.

Objektově orientované programovací jazyky jsou označovány jako jazyky „třípůlté“ generace. Přináší sebou několik užitečných technik, které usnadňují návrh programu, redukují redundance zdrojových kodů, přináší efektivnější rozšiřitelnost vyvíjených *API* a v konečném důsledku i lepší přehlednost vyvíjených kódů. To vše za cenu určité ztráty početního výkonu. *Objektově orientované programování* (dále *OOP*) je někdy označováno jako programátorské paradigma, neboť popisuje nejen způsob vývoje a zápisu programu, ale i způsob, jakým návrhář programu o problému přemýšlí. Při vývoji rozsáhlejších projektů je efektivní využít právě *OOP*. Protože jazyk *JS* podporuje *OOP*, využívám jej v této práci. Existuje však řada protichůdných názorů, zda se jedná o kvalitní nástroj z hlediska *OOP*. V následujícím textu přiblížím principy *OOP* a předvedu správný způsob, jak je využít v *JS*.

2.2 Čistý a hybridní jazyk

Základním hlediskem, podle kterého se dělí *OOP* jazyky je tzv. *objektová čistota* vyvíjeného kódu. Řada jazyků s *OOP* vznikla jako rozšíření některého jazyku třetí generace (např. *Object Pascal*, *C++*, nebo *Objective-C*). Je zde možnost psát kód v daném jazyce bez *OOP* nebo obě techniky libovolně kombinovat. I kdybychom se však snažili v těchto jazycích psát *objektově čistý* kód sebevíc, nikdy by se nám to nepodařilo. Vždy by zde existovala část zdrojového kódu, která by nebyla tvořena objektem. Příkladem je funkce `main()` v jazyce *C++*, kterou musí obsahovat

každý program a nemůže nikdy tvořit část objektu. Tyto jazyky, které nevytváří *objektově čistý*, kód se nazývají jazyky *hybridní* a patří mezi ně i *JS*.

Opakem *hybridních* jsou *čisté* jazyky. V nich nelze napsat program, který by nebyl objektově *čistý*. Patří mezi ně populární jazyk *Java*. Čisté jazyky se nehodí pro vývoj malých aplikací, pro které je *JS* určen a zřejmě proto není *JS* čistým jazykem. Dají se ale pomocí *OOP* v *JS* psát efektivně i středně velké a větší projekty?

2.3 Objekty, jejich vlastnosti a realizace v JavaScriptu

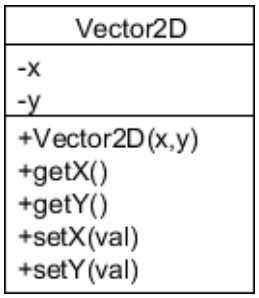
Aby byl programovací jazyk *objektový*, musí umožnit programátorovi definovat *objekty*, vytvářet jejich *instance* a využívat *dědičnost* a *polymorfismus*. Definicí *objektu* je *třída* (angl. *class*) a ta na základě svého *konstruktoru* (konstrukční funkce) umožňuje vytváření *instancí*. Přítomnost *konstruktoru* je jednou z vlastností, kterou se *objekty* liší od prostých proměnných. *Dědičnost* a *polymorfismus* jsou základní prvky *OOP* a budu se jim věnovat podrobněji v dalších částech této práce.

Při návrhu objektového kódu se často využívá tzv. *diagram tříd*. Jedná se o grafickou reprezentaci určitých *tříd* a určitých vztahů mezi nimi tak, že jsou v něm prvky *OOP* vyznačeny podle standardů definovaných jazykem *UML* (Unified Modelling Language). V následujících kapitolách se budu věnovat jednotlivým metodikám *OOP* a ukáži, jak je lze zaznamenat pomocí *diagramu tříd*. Později v práci budou ukázány *UML* diagramy navržené pro vyvíjený *XML* editor.

2.3.1 Třídy a zapouzdření

Objekt je úložištěm dat, ke kterým je přistupováno pomocí *instance*. Data v rámci *třídy* se nazývají vlastnosti (*atributy*, angl. *properties*). Bývá zvykem proměnné v objektu „zapouzdřit“, tzn. deklarovat je tak, aby k nim nebylo možné přistupovat mimo *třídu*. *Třída* tyto „zapouzdřené“ položky obvykle zpřístupní pomocí *metod*. *Metody* (*členské funkce*) jsou *funkce*, které tvoří součást *objektu*.

„Zapouzdřené“ *atributy* tvoří tzv. *soukromé rozhraní*. To je skryto před okolím (*objektem* přistupujícím k této *instanci* — *klientem*). *Soukromé* položky si *objekt* spravuje sám a nemůže se stát, že by např. *klient* jejich přímou změnou narušil správnou činnost *instance*. Pomocí *veřejného rozhraní objekt* určí, které *soukromé atributy* může *klient* měnit (případně jaké vedlejší operace změna vyvolá), které budou k dispozici pouze pro čtení a které budou před *klientem* úplně schovány.



Obrázek 1: Diagram třídy Vector2D

V *diagramu tříd* jsou *třídy* vyobrazeny jako entity obsahující tři části. V první je název *třídy*, ve druhé její *atributy* tvořené prostými proměnnými a ve třetí se zapisují *metody*. Prefix plus resp. mínus znamená zda je položka *veřejná* resp. *soukromá*. Na obr. 1 je diagram jednoduché třídy `Vector2D`. *Konstruktor* je zaznamenán jako *metoda* s názvem *třídy*.

Kompilované programovací jazyky bývají běžně *statické* — při *deklaraci proměnných* pevně přiřazují *datový typ*. Každá *třída* tvoří nový datový typ, takže při deklaraci nových *tříd* programátor deklaruje nové *datové typy*. Naproti tomu *dynamické* jazyky (překládané zpravidla *interpretem* za běhu) přistupují k *datovým typům* daleko svobodněji a proměnné deklarují bez nich. *JS* je *dynamickým* jazykem, nepřiřazuje typy v deklaraci proměnných a nelze v něm definovat *třídy* tak jak je zvykem např. v jazycích *C++* nebo *Java*. Možná právě proto nezkušení vývojáři často neví, že *JS* patří mezi *objektové* jazyky.

2.3.2 Třídy v JavaScriptu

JS nemá klasické *třídy* a i proto je *OOP* v tomto jazyku trochu neobvyklé. Existují názory, že *třídy* v *JS* nejsou pravými *třídami*, ale opak je pravdou. *JS třídy* totiž splňují definici, že *třída* je konstrukt, který je využíván jako předpis sloužící k vytváření *objektů* této *třídy*. Za *třídu* v *JS* považujeme samotný *konstruktor*, který využívá vlastnosti *prototype*.

V *JS* vše, co není primitivním datovým typem, je *objektem*. Např. funkce, pole i regulární výrazy jsou *objekty*. Všechny *třídy* v *JS* jsou odvozeny od datového typu `Object`, který tvoří vrchol hierarchie *tříd*.

Funkce, *metody*, *konstruktory* i *třídy* jsou v *JS* jednou a tou samou funkcí. Pojmenování funkce určuje roli, kterou funkce hraje. Funkce je *prostou funkcí*, pokud netvoří součást žádného *objektu* a *metodou* pokud součástí *objektu* je. *Třída* a *konstruktor* je v *JS* jedno a to samé, pro vytvoření nové *třídy* v *JS* je zkrátka třeba definovat *konstruktor*, čili funkci. Možností jak v *JS* realizovat

```
1     /* class Vector2D */
2     Vector2D = function(x,y) {
3         /* private properties */
4         var x = x;
5         var y = y;
6         /* public properties */
7         this.getX = function() {
8             return x;
9         }
10        this.getY = function() {
11            return y;
12        }
13        this.setX = function(val) {
14            x = val;
15        }
16        this.setY = function(val) {
17            y = val;
18        }
19    }
```

Obrázek 2: Třída Vector2D v JavaScriptu

zapouzdření a jak deklarovat *veřejné metody* je více. Většina postupů nějakým způsobem využívá klíčové slovo `this`, to pomocí operátoru tečka zpřístupňuje *kontext* — neboli funkci resp. *objekt* v kterém je použito. Pokud se `this` nenachází uvnitř žádné funkce (*objektu*), odkazuje na *objekt* `window`. Na obr. 2 je ukázáno, jak lze v *JS* definovat *třídu* `Vector2D` z předchozí ukázky[1]. Přestože tento způsob není úplně správný, uvádím jej zde, protože je velmi rozšířený. Z hlediska správného *OOP* v *JS* je však nešťastnou ukázkou, jak se to dělat nemá.

Jediným správným způsobem jak v *JS* definovat rozhraní *třídy* je využitím vlastnosti jménem `prototype` tak, jak je ukázáno na obr. 3. Vlastnost `prototype` má každá funkce a její výchozí hodnotou je prázdný *objekt*. Tato deklarace je přirozená a plyne z návrhu jazyka. Protože je *JS dynamický jazyk*, lze jeho *třídy* měnit za běhu. Jakákoliv změna *třídy* za běhu se musí projevit i ve všech jejích *instancích*. Toho lze docílit pouze použitím `prototype`, je to totiž *vlastnost* sdílená všemi *instancemi* jedné *třídy*. V ukázce na obr. 3 nejsou možné případné další modifikace *třídy* za běhu a problematická by byla i správná realizace *dědičnosti*.

2.3.3 Práce s instancemi

Instance jsou vytvářeny voláním *konstruktoru*. Po alokaci lze pracovat s *veřejným rozhraním objektu* (dle potřeb vyvíjeného programu) a kteroukoliv instanci lze smazat (uvolnit z paměti počítače). Při uvolňování *instance* je volán *destruktor*, neboli *metoda* volaná těsně před okamžikem, kdy *instance* přestane fyzicky existovat.

[1] Autorem této metody je Douglas Crockford.

```
1       /* class Vector2D */
2       Vector2D = function(x,y) {
3           /* private properties */
4           this.x = x;
5           this.y = y;
6       }
7       Vector2D.prototype = {
8           /* public properties */
9           getX: = function() {
10              return this.x;
11          },
12          getY = function() {
13              return this.y;
14          },
15          setX = function(val) {
16              this.x = val;
17          },
18          setY = function(val) {
19              this.y = val;
20          }
21      }
```

Obrázek 3: Třída Vector2D v jazyce JavaScript pomocí vlastnosti prototype

Obecně může každá *třída* obsahovat více *konstruktorů* a programátor tak může mezi nimi vybírat a provádět tak různé inicializace stejné *třídy*. *Destrukor* je však v *třídě* vždy maximálně jeden. Jak již bylo řečeno, *vlastnosti objektu* jsou přístupné pomocí *instance*, výjimkou jsou tzv. *statické vlastnosti*, které jsou společné všem *instancím* a jsou přístupné pomocí *třídy*. Lze je volat i když neexistuje žádná *instance* dané *třídy*, existují vždy právě jednou nehledě na počtu *instancí*.

```
1       v1 = new Vector2D(0,0);
2       v1.setY(10);
3       alert( 'The Y value is: ' + v1.getY() );
4       delete Vector2D;
```

Obrázek 4: Instance Vector2D v jazyce JavaScript

V *JS* nelze definovat *třídy* s více jak s jedním *konstruktorem* a nelze jim přiřadit *destruktor*. Alokace nových *instancí* se v *JS* provádí pomocí operátoru `new` a dealokace pomocí operátoru `delete`. V ukázce na obr. 4 je vytvoření *instance* `Vector2D`, ukázkové použití jejího veřejného rozhraní voláním *metod* definovaných v této *třídě* a následná dealokace *objektu*. *Instance* není nutné ručně dealokovat, *JS* obsahuje tzv. *garbage collector*, který automaticky uklízí *objekty*, které nejsou nadále využívány. Avšak ruční dealokace pomocí `delete` může ušetřit paměť a zefektivnit tak výsledný skript.

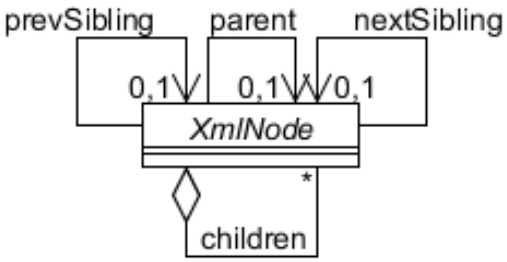
2.3.4 Vazby mezi objekty a zasílání zpráv

Odkazy na *objekty* se v *JS* realizují pomocí tzv. *referencí*. *Reference*, neboli *referenční proměnná*, je proměnná, která nevytváří novou proměnnou jako takovou, ale slouží jako alias jiné proměnné resp. *instance*. *Reference* jsou velmi důležitou součástí jazyka *JS*, *JS* totiž pracuje s *objekty*, o jejichž alokaci se stará sám prohlížeč, který je zpřístupňuje skriptům právě pomocí *referencí*. *Referenční proměnná* se v *JS* deklaruje obdobně jako běžná proměnná s vynecháním klíčového slova `var`.

Objekty bývají zpravidla obklopeny jinými *objekty*, kterým poskytují své služby, případně po nich služby požadují. Okolní *objekty* pracují s *objektem* pomoci jeho *metod* a tomuto procesu se někdy říká *zasílání zpráv mezi objekty*. Hlavička (deklarace) *metody* totiž definuje jakýsi komunikační protokol, pomocí kterého se *objekt* dorozumívá s okolním světem. To znamená, že *reference* v jednom *objektu* na jiný *objekt* poskytuje *objektu* s *referencí* možnost využívat druhý *objekt*. Obecná vazba mezi *objekty* se nazývá *asociace*, a v tomto případě jde o *prostou asociaci*. Během návrhu tedy vývojář obvykle člení konkrétní problematiku do jednodušších celků — *objektů* a vazby mezi nimi zaznamenává a vytváří tak diagramy *UML*. *Prostá asociace* se zapisuje jako souvislá hrana, která může a nemusí být ohodnocena tak, aby vystihovala danou problematiku (její hodnota záleží na návrháři). Směr hrany vyjadřuje směr *asociace*, např. pokud třída *A* asociuje třídu *B*, směřuje i hrana od třídy *A* k třídě *B*. Dále zde může návrhář zaznamenat, kolik *asociací* vazba vytváří. Symbol * znamená libovolné množství, dalšími možnostmi jsou např.: 1 (právě jedna); 5, 10, 20 (5, 10, nebo 20); 1..3 (1 až 3); 8..* (8 a více) atp.

Dalšími druhy *asociace* jsou *agregace* a *kompozice* (neboli *kompozitní agregace*). Ty bývají zaznamenány pomocí hran zakončených kosočtvercem (ten je na straně celku), u *kompozice* je kosočtverec vyplněn. Opět je zde možnost vyčíslit množství *asociace* obdobně jako u *prosté asociace*. *Agregace* je speciální formou *asociace*, která specifikuje vztah mezi agregujícím (celkem) a jeho konstituentem (částí). Jeden agregovaný *objekt* může být potenciálně částí více *objektů*. *Kompozitní agregace* je silnější formou, která vyžaduje, aby jakákoliv *instance* konstituenta byla součástí nejvýše jednoho agregujícího celku. Ten pokud je smazán, smaže i všechny své části *kompozice*. Specifikace *UML* poměrně přísně definuje jednotlivé druhy *asociací*, v praxi k nim však vývojáři obvykle nepřistupují tak striktně. Např. mnoho návrhů nerozlišuje mezi *agregací* a *kompozicí*, navíc jsem se párkrát setkal se stejnými *návrhovými vzory* (standardními postupy návrhu objektové aplikace) vyjádřenými pokaždé odlišnými *asociacemi*. V praxi zkrátka záleží především na citu konkrétního návrháře, se kterým k dané problematice přistupuje. Je zřejmé, že

o alokaci konstituentů se stará *konstruktor objektu*, který představuje celek.



Obrázek 5: Třída XmlNode a její asociace

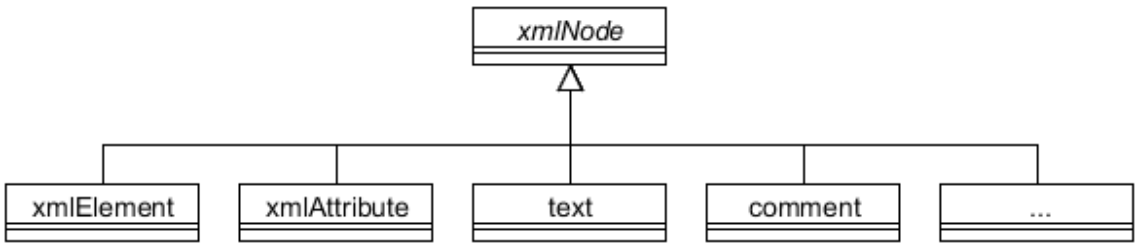
Jednotlivé druhy *asociací* jsou předvedeny v ukázce na obr. 5, kde je návrh *třídy*, která reprezentuje větev stromu *XML* dokumentu. Ta asociuje své sousední a nadřazené větve (*prevSibling*, *nextSibling* a *parent*) a agreguje podřízené větve (*children*). V tomto návrhu lze libovolně procházet celý strom *XML* na základě *reference* na kteroukoliv jeho větev. Obdobný návrh využívá i *DOM* (objektové *API* pro *XML*, viz 3.2).

2.3.5 Princip abstrakce

Abstrakce v *OOP* úzce souvisí s pojmy *dědičnost* a *polymorfismus*. Díky těmto technikám, může programátor přihlížet na určitou problematiku v určité *abstraktní* rovině, kde neexistují konkrétní implementace. Při návrhu se tak lze snadno oprostit od detailů, které nejsou v dané situaci důležité.

Třída `XmlNode` z předchozí ukázky je *abstraktní*, proto je její název vyznačen kurzívou. Tato *třída* definuje *asociace* resp. vztahy mezi okolními větvemi, představuje však obecnou větev *XML* stromu formou obecného rozhraní pro konkrétní typy *XML* větví. Jednotlivé *XML* větve mohou být různého druhu, ale vlastnosti *třídy* `XmlNode` budou společné všem, nehledě na to jestli jde o větev typu *element* či větev typu *komentář* atp. Vlastnosti *tříd*, které jsou definované, ale nemají konkrétní implementaci, se nazývají *abstraktní vlastnosti* a každá *třída* obsahující nějakou *abstraktní vlastnost* je *abstraktní třídou*.

Dědičnost (*generalizace*), je hierarchický vztah, kdy nová *třída* představující potomka (angl. *subclass*) obsahuje všechny *vlastnosti třídy* představující jejího předka (angl. *superclass*). Předek bývá někdy označován jako *bázová třída* (angl. *base class*). *Generalizace* se v *diagramu tříd* vyznačí orientovanou hranou zakončenou trojúhelníkem, směřujícím k *třídě* předka. V případě třídy `XmlNode` bude tato *třída* představovat *bázovou třídu* pro *třídy* tvořící konkrétní typy *XML* větví (`XmlElement`, *Comment*...), což vyjadřuje *diagram tříd* na obr. 6 (nejsou zde zachyceny všechny druhy *XML* větví, protože účel kapitoly není podrobný rozbor jazyka *XML*).



Obrázek 6: Dědičnost třídy xmlNode

Potomek může krom zděděných *vlastností* obsahovat některé své specifické *vlastnosti* či vlastnosti předka měnit neboli překrývat. Pokud chceme používat *instanci* potomka *abstraktní bázové třídy*, musí potomek překrýt všechny *abstraktní vlastnosti* předka neabstraktními, tj. doplnit chybějící implementace předka. Stejné rozhraní tak v různých dědicích vyvolá různé operace lišící se konkrétní implementací. To, že se odkazovaný *objekt* chová podle toho, jaké *třídy* je *instancí*, je právě principem *polymorfismu*. Statické programovací jazyky využívají tzv. *polymorfismus podmíněný dědičností*. To znamená, že na místo, kde je očekávána *instance* nějaké *třídy*, můžeme dosadit i *instanci* libovolného jejího dědice, neboť rozhraní *třídy* je podmnožinou rozhraní dědice. Dynamické jazyky (jako je i *JS*) využívají jednodušší *polymorfismus nepodmíněný dědičností*, kde je dostačující, jestliže se rozhraní (nebo jejich požadované části) u různých *tříd* shodují, avšak jejich implementace se liší — pak jsou vzájemně *polymorfní*.

2.3.6 Dědičnost a polymorfismus v JavaScriptu

Ačkoliv má *JS* určité prvky *dědičnosti*, nejdená se o klasickou *dědičnost*, čili *dědičnost* na základě *tříd* (angl. *classical inheritance*) známou ze statických jazyků. Jde spíše o jakousi simulaci, protože *JS* nepodporuje zápis *tříd* (umožňuje pouze zápis *konstruktorů*). *Dědičnost* je v *JS* tzv. prototypová (angl. *prototypal inheritance*). Klíčový prostředek *dědičnosti* v *JS* je *vlastnost* `prototype` zmíněná v kapitole 2.3.2.

Třída `Vector2D` z kapitoly 2.3.1 by mohla sloužit jako předek pro třídu `Vector3D`, která třídu `Vector2D` rozšíří o zetovou souřadnici. Použitím `prototype` to lze udělat obdobně jako v ukázce na obr. 7. Tuto techniku pro realizaci dědičnosti uvádí mnoho učebnic a článků o *JS*. Přestože je toto doporučení funkční, nepoužil jsem ho z důvodů uvedených v následujícím odstavci.

Takto definovaná *třída* `Vector3D` inicializuje *privátní vlastnosti* `x` a `y` zděděné od *třídy* `Vector2D`. Ty jsou ale také inicializovány už v *konstruktoru* `Vector2D`, jejich inicializace jsou

```
1     /* class Vector3D */
2     Vector3D = function(x,y,z) {
3         /* private properties */
4         this.x = x;
5         this.y = y;
6         this.z = z;
7     }
8     /* inherit from Vector2D */
9     Vector3D.prototype = new Vector2D();
10    /* public properties */
11    Vector3D.prototype.getZ = function() {
12        return this.z;
13    }
14    Vector3D.prototype.setZ = function(val) {
15        this.z = val;
16    }
```

Obrázek 7: Dědičnost v JavaScriptu

záležitostí této *třídy* a je neefektivní dělat to znovu v *konstruktoru* `Vector3D`. *Vlastnosti* `x` a `y` jsou zásadní pro obě *třídy*, jsou inicializovány v *konstruktoru*, aby mohli mít platné hodnoty hned po vytvoření *instance*. Aby to ale platilo i při vytváření *instancí* `Vector3D`, musí se o inicializaci *vlastností třídy* `Vector2D` postarat *konstruktor* `Vector3D`, musíme tedy psát znovu stejný kód a vytváříme tak zbytečné redundance. Problém je v tom, že voláme *konstruktor předka* během deklarace *třídy potomka* (řádek 9 na obr. 7). To odporuje zásadě *OOP*, že *konstruktor třídy* je volán při vytváření *instance*, nikoliv během deklarace *třídy*. Pokud bychom volali *konstruktor* `třídy Vector2D` v konstruktoru *třídy* `Vector3D`, nemusela by *třída* `Vector3D` provádět znova inicializace *vlastností* `Vector2D`. Navíc v případě, že by *konstruktor* předka obsahoval důležité alokace dalších např. *kompozitních objektů*, při použiti techniky z ukázky na obr. 7 by se tyto *objekty* alokovaly už během návrhu *třídy*, což je zcela špatně.

```
1     /* helper function for inheritance */
2     inherit = function(child,parent) {
3         /* helper temporary constructor */
4         var foo = function() {};
5         foo.prototype = parent.prototype;
6         child.prototype = new foo();
7         /* convention for calling overrided methods */
8         child.superClass = parent.prototype;
9         /* constructor references it's instance */
10        child.prototype.constructor = child;
11    }
```

Obrázek 8: Pomocná funkce pro dědění v JavaScriptu

Pro správnou realizaci dědičnosti v *JS* je potřeba funkce `inherit`[2]. Ta netvoří standardní součást jazyka *JS*, proto je její implementace uvedena na obr. 8. Nejprve je vytvořen pomocný

[2]Mimochodem naprosto stejnou techniku používá i *JavaScriptová* knihovna *Google Closure* od Googlu.

```
1       /* class Vector3D */
2       Vector3D = function((x,y,z) {
3           /* call base constructor */
4           Vector2D.call(this,x,y);
5           /* private properties */
6           this.z = z;
7       }
8       /* inherit form Vector2D */
9       inherit(Vector3D,Vector2D);
10      /* public properties */
11      Vector3D.prototype.getZ = function() {
12          return this.z;
13      }
14      Vector3D.prototype.setZ = function(val) {
15          this.z = val;
16      }
```

Obrázek 9: Dědičnost v JavaScriptu pomocí funkce inherit

dočasný *konstruktor*, pomocí kterého je vytvořen `prototype` *potomka* a obejde se tak nutnost volat konstruktor předka. `Prototype` *předka* si *potomek* udržuje ve *vlastnosti* `superClass`, což je užitečné zejména v případě, kdy *potomek* překrývá některou metodu *předka* a v rámci její nové implementace potřebuje volat *metodu*, kterou překrývá (může tak např. překrytím rozšířit původní *metodu* o novou funkcionalitu, namísto jejího úplného překrytí).

Třída `Vector3D` odvozená pomocí funkce `inherit` je na obr. 9. Je zde vidět, že *třída* neobsahuje předchozí redundance, volá *konstruktor* `Vector2D` ve svém *konstruktoru* (řádek 4 v ukázce) a ne během dědění.

V kapitole 2.3.5 jsem zmínil, že *abstraktní třída* `XmlNode` slouží jako předek pro konkrétní druhy *XML* větví. Zatím jsem ale neuvedl žádnou její *abstraktní metodu*. Mohla by jí být např. *metoda* pro tzv. *serializaci XML* větve, tj. převod na textový řetězec. Jednotlivé druhy *XML* větví totiž mohou být *serializovány* různě. V poslední ukázce kapitoly věnované *OOP* v *JS* na obr. 10 uvádím, jak by mohla vypadat třída *xmlNode* s *abstraktní metodou* `serialize` a její odvozené třídy `XmlAttrbute` a `Comment` implementující tuto *metodu*. V *JS* neexistují *abstraktní třídy* jako takové, protože nelze vytvářet *metody* bez implementace. *Abstraktní metoda* je určena pouze pro překrývání a neměla by být nikdy volána v rámci *abstraktní třídy*, ale pouze v rámci *potomků*, kteří ji překrývají. Opět neexistuje konkrétní řešení, jak vytvářet v *JS abstraktní třídy*. Moje řešení je vyvolání *vyjímky* (angl. *exception*), která upozorní programátora na to, že volá *abstraktní metodu*.

```
1       /* helper function for inheritance */
2       inherit = function(child,parent) {
3           /* helper temporary constructor */
4           var foo = function() {};
5           foo.prototype = parent.prototype;
6           child.prototype = new foo();
7           /* convention for calling overrided methods */
8           child.superClass = parent.prototype;
9           /* constructor references it's instance */
10          child.prototype.constructor = child;
11      }
12
13      /* class XmlNode*/
14      XmlNode = function(name,val)  {
15          /* private properties */
16          this.name = name;
17          this.val = val;
18      }
19      XmlNode.prototype = {
20          /* public properties */
21          serialize: function() {
22              throw 'Abstract method called!';
23          }
24      }
25
26      /* class XmlAttribute */
27      XmlAttribute = function(name,val) {
28          /* call base constructor */
29          XmlNode.call(this,name,val);
30      }
31      /* inherit from XmlNode */
32      inherit(XmlAttribute,XmlNode);
33      /* override serialize method */
34      XmlAttribute.prototype.serialize = function() {
35          return this.name + '=' + this.val;
36      }
37
38      /* class Comment */
39      Comment = function(val) {
40          /* call base constructor */
41          XmlNode.call(this,'',val);
42      }
43      /* inherit from XmlNode */
44      inherit(Comment,XmlNode);
45      /* override serialize method */
46      Comment.prototype.serialize = function() {
47          return this.val;
48      }
```

Obrázek 10: Polymorfismus v JavaScriptu

2.4 Zhodnocení přístupu k JavaScriptu

JS je často kritizovaným jazykem, ať už kvůli neustálým potřebám optimalizace, nebo z důvodů jeho implementace *OOP*. *JS* není špatný jazyk pro *OOP*, záleží spíš na přístupu vývojáře. Je to *dynamický* jazyk a právě dynamika je jeho silnou stránkou. Dává programátorovi určitou svobodu, kterou by v jiných jazycích neměl, díky čemuž může např. měnit *třídy* za běhu, což je prvek u statických (kompilovaných) jazycích nevídaný. Díky této svobodě pak ale mohou programátoři realizovat stejné prvky *OOP* různými způsoby a vytvářet tak odlišná řešení, která jsou sice na první pohled funkční, ale při podrobnějším rozboru selhávají.

Dynamika *JS* může také vést k mnohým chybám, protože programátor *JS* nikdy nemá jistotu, že jiný programátor nějakou jeho proměnnou nezmění resp. nepředefinuje svým skriptem, ať už je tato proměnná sebevíc zapouzdřena. Změna této proměnné může nakonec způsobit, že původní skript přestane zcela fungovat a zdá se, že je chyba na straně programátora tohoto skriptu, ačkoliv ve skutečnosti ji způsobil někdo jiný.

V této kapitole jsem ukázal, že existuje způsob, jakým lze v *JS* správně vytvářet *třídy*, implementovat *dědičnost* a s ní spjatý *polymorfismus*. Pomocí ukázaných technik není problém z *JS* udělat korektní prostředek pro *OOP* a nakonec to znamená, že přestože je *JS* původně navržen pro krátké jednorázové skripty, hodí se i pro vývoj větších aplikací. Řekl bych že to je jeden z hlavních důvodů, proč je *JS* stále rozšířený, i když jeho konkurence stále sílí (viz poměrně nové rozhraní *Silverlight* od Microsoftu). Webový prohlížeč je dnes chápán spíše jako rozhraní pro webové aplikace, než jako prostý prostředek pro prohlížení *HTML* dokumentů, vývoj skriptů už proto dávno není jen o „*oživení*“ stránek pomocí dynamického HTML (*DHTML*), ale o vývoji aplikací jako takových, proto je úloha *OOP* v rámci webových aplikací stále více důležitá.

Největší problémy způsobují webové prohlížeče, které *JS* zprostředkovávají. Nutnost optimalizace mnohdy dělá z poměrně jednoduchých zadání zbytečně složitá řešení, roste tak velikost skriptů, jejich přehlednost klesá a především roste časová náročnost spojená s vývojem takového skriptu. Další slabinou *JS* jsou „*mystifikace*“ a omyly spojené s tímto jazykem. Se správným přístupem všeobecně uznávané doporučené publikace [1] se z *JS* stává účinný nástroj.

3 Vývoj JavaScriptového XML editoru

Vývoj *XML* editoru, je poměrně rozsáhlé téma a nabádá k diskusi, co všechno by editor mohl nabízet. Než jsem začal se samotným návrhem editoru, přemýšlel jsem, které funkce do aplikace implementuji. Ty uvádím v následujícím odstavci, jejich realizace bude postupně probrána v této kapitole.

Režim zdrojového kódu – v tomto režimu je zobrazeno zdrojové *XML* dokumentu a jeho editací je možné jej libovolně upravovat obdobně jako např. v *Poznámkovém bloku* v prostředí Windows. Je důležitý zejména v případě, kdy uživatel chce do dokumentu přidat nějaký element, který v uživatelském prostředí editoru není přítomen. Tento postup editace *XML* je asi nejméně pohodlný a vyžaduje po uživateli určitou odbornost.

Editace elementů a atributů – je zřejmé, že hlavním účelem editoru je změna editovaného kontextu. V případě *XML* jde o přidávání, resp. mazání elementů a editace jejich obsahu.

Zvýrazňování syntaxe – pro lepší přehlednost zdrojového *XML* snad každý editor nabízí možnost zvýrazňování syntaxe v režimu zdrojového kódu (ačkoliv webové editory jsou výjimkou).

Kontrola validity – uživatelské rozhraní editoru by mělo umožňovat uživateli přidávat na určité pozice takové elementy, které jsou v daném kontextu platné. Aplikace by měla zabránit operacím, jejichž výsledkem by byl nevalidní dokument.

Načítaní a ukládání dokumentu – samozřejmostí je možnost uložení a opětovné načtení vyvíjených dokumentů.

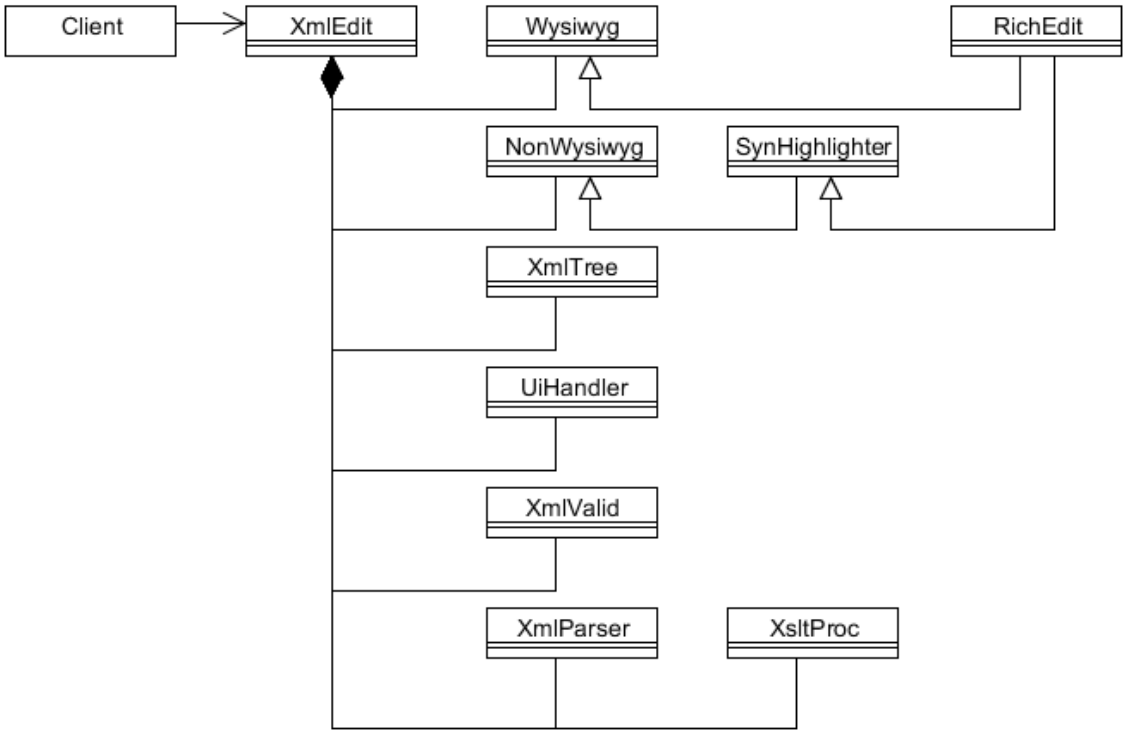
Režim rozbalovacího stromu – jde o režim, ve kterém je dokument zobrazen jako rozbalovací strom, který je přehlednější než samotný zdrojový kód. Pro člověka, který se s *XML* teprve seznamuje, má tento režim názorný význam.

Podpora jmenných prostorů – v jazyce *XML* je možné kombinovat několik jazyků do jednoho dokumentu pomocí tzv. jmenných prostorů. I s tím by měl editor počítat např. formou dynamického uživatelského rozhraní, které se aktualizuje na základě právě aktivního jmenného prostoru.

WYSIWYG režim – můj editor navíc nabízí *WYSIWYG* režim, který umožňuje editaci *XML* bez znalosti konkrétního *XML* jazyka a dokonce i bez znalosti jazyka XML jako takového. Je to důležitý prvek jakéhokoliv editoru, který existuje na webové stránce. Protože vyvíjím webový editor, lze v něm takto psát pouze XML pro takové jazyky, které lze převést na HTML ekvivalent. V opačném případě, lze použít editor bez WYSIWYG režimu.

3.1 Návrh aplikace

Framework *jQuery*, který využívám v tomto projektu, nabízí řadu užitečných funkcí. Jednou z nich je pohodlný vývoj zásuvných modulů (*pluginů*) libovolně rozšiřující *jQuery* o novou funkcionalitu. Psaní *pluginů* v *jQuery* je velmi snadné a pohodlné, stačí rozšířit objekt představující *jQuery* (ten je přístupný pomocí funkce $) o metodu tvořící nový *plugin*. V případě *OOP* při vývoji *jQuery pluginů* definuje každý nový *plugin* novou třídu a uvnitř inicializační metody *pluginu* je vytvořena instance této třídy.



Obrázek 11: Diagram tříd XML editoru

Instance tříd navržené pro účely mého *XML* editoru spolupracují prostřednictvím kořenového objektu `xmlEdit`. Jde o kompozitní objekt spravující jednotlivé části editoru. Jakmile je in-

stanciován objekt `xmlEdit`, jsou také automaticky inicializovány ostatní *pluginy* reprezentující podřízené části kompozice `xmlEditu`. Jednotlivé *pluginy* lze použít samostatně bez `xmlEditu` v jiných JS aplikacích. Další výhodou tohoto návrhu je pohodlná orientace ve zdrojových skriptech, ladění a rozšířitelnost, protože jednotlivé úpravy jsou prováděny na úrovni jednotlivých modulů, nezávisle na zbytku aplikace.

3.2 Zpracování dat formátu XML

Při vývoji aplikací, které jakkoliv pracují s *XML* je důležité vědět, jak zpracovávat data formátu *XML* pomocí programového rozhraní dostupného v daném programovacím jazyce. Nejprostější možnost je zpracovávat *XML* jako textový řetězec a využít dostupné *API* pro práci s textem (v případě JS např. regulární výrazy, nebo metody, které nabízí objekt `String`). Někdy je tento přístup postačující, ale mnohé operace jsou v něm obtížně realizovatelné a neefektivní.

Proto existují *API* specializované přímo pro práci s *XML*, jsou to *SAX*, ve kterém programátor pomocí tzv. *callback* funkcí reaguje na určité události *XML* dokumentu a o něco pomalejší avšak flexibilnější *DOM*, který je v podstatě odrazem *XML* na poli OOP.

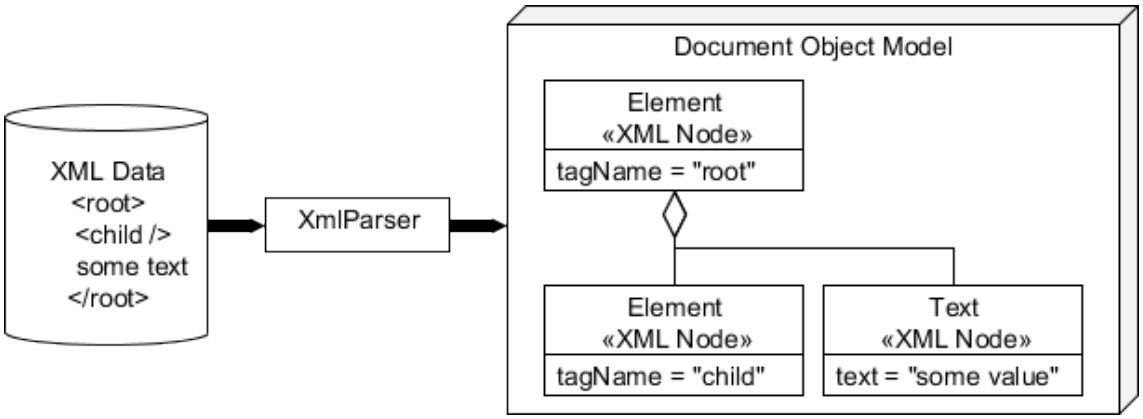
DOM byl standardizován *W3C* a lze ho použít i v případě *HTML* dokumentů, proto ho webové prohlížeče nativně podporují a v případě *XML* v JS se využívá právě *DOM*. Z hlediska optimalizace je velkým problémem fakt, že různé prohlížeče implementují různé části *DOM* a dokonce někdy stejné vlastnosti zpřístupňují pod různými symbolickými názvy. Cílem různých JS frameworků včetně *jQuery* často bývá mimo jiné i řešení těchto nekompatibilit.

DOM je robustní rozhraní a proto je rozděleno do několik částí (tzv. *DOM levels*), často webovým vývojářům stačí pouze jeho základní část (*DOM level 1*), kterou implementují snad všechny webové prohlížeče. V ní je možné libovolně procházet dokument a modifikovat jeho obsah. *DOM* dokument zabere v paměti počítače víc než samotný zdrojový kód dokumentu, avšak jednotlivé operace s dokumentem jsou daleko efektivnější a rychlejší, než kdyby se procházel zdrojový kód reprezentovaný textovým řetězcem. Představte si např., že se potřebujete odkázat na textový obsah dokumentu. V případě textového řetězce, i za použití regulárních výrazů budou výsledkem atomické operace procházející řetězec znak po znaku. Při použití *DOM* se stačí odkázat na jednu adresu v paměti počitače. Na druhou stranu, pokud chcete vytvořit nový element, bývá to zpravidla jednoduché realizovat pomocí textového řetězce. Stačí v něm mít uložený jeho zdrojový kód, nicméně např. vytvoření jednoduchého odstavce s jednou textovou větví vydá v *DOM* na několik řádků kódu a výsledek není zrovna přehledný.

Framework jQuery se dá z určitého hlediska chápat jako nadstavba *DOM*. Metody *jQuery* pro manipulaci dokumentu připomínají *DOM*, dokonce ho rozšiřují o některé nové funkce a řeší některé nekompatibility. Velikou výhodou také je, že lze v *jQuery* libovolně kombinovat textové řetězce představující dokument nebo *CSS* selektor s rozhraním *DOM*. Je tak možné např. vložit do dokumentu nový element zadaný textovým řetězcem a následně se na něj odkazovat pomocí *jQuery* nebo přímo *DOM API*. Na druhou stranu *jQuery* jako framework pro JS se zaměřuje na jazyk *HTML*, přestože v něm lze jednotně převádět *DOM* dokument v *HTML* na řetězec a naopak, v případě *XML* to nelze.

3.2.1 Parserování XML do Document Object Modelu

Protože *XML* data představující kontext editoru zpracovávám v JS pomocí *DOM*, stěžejním atributem objektu *XMLedit* je reference na *XML DOM*. Ten představuje objektový obraz editovaného *XML* dokumentu a je tvořen instancemi objektů, které reprezentují jednotlivé *XML* větve (tzv. *DOM nodes*). Větve jsou na sebe navázány pomocí asociací tak, že ve výsledku vytváří celý strom *XML*, pro názornost uvádím jednoduchý příklad na obr. 12. V kombinaci s *jQuery* lze s takovým *XML* stromem velmi efektivně pracovat (výhodou je možnost využití *CSS* selektorů), aby to však bylo možné, je důležité instance *DOM* dokumentu nejprve vytvořit.



Obrázek 12: Document Object Model (zdroj: vlastní)

O převod *XML* (resp. *HTML*) dat ze zdrojového kódu do *DOM* se stará *parser*. Je to program (případně objekt), který obecně provádí analýzu dat textového formátu. *XML parsery* kontrolují validitu (viz kapitola 3.2.2) a v případě JS je jeho výstupem *DOM* resp. instance objektu *DOM* dokument. Ačkoliv zde existují drobné rozdíly mezi objekty *XML DOM* dokument a *HTML*

DOM dokument, až na drobné odlišnosti je jejich rozhraní totožné (tvoří součast specifikace *W3C*). Jazyk JS využívá *XML parser* zabudovaný ve webovém prohlížeči. *IE* využívá své vlastní nestandardní rozhraní, takže skript využívající *XML parser* musí být optimalizovaný, aby byl přenositelný.

Úlohy parserování *XML* do *DOM* a jeho zpětná serializace jsou zásadní pro činnost některých částí mého editoru. K tomu je využíván objekt `XmlParser`, který jsem navrhl tak, aby byl co možná nejflexibilnější. Stejně jako jiné části mé aplikaci, lze jej použít samostatně bez `XmlEdit` kdekoliv, kde je potřeba pracovat s *XML* v JS.

```
1     urlToDom = function(url) {
2         if ( navigator.appName.toLowerCase() != 'microsoft internet explorer' ) {
3             xhttp = new XMLHttpRequest();
4             xhttp.open('GET',url,false);
5             xhttp.send('');
6             var xmlDoc = xhttp.responseXML;
7         }
8         else {
9             var xmlDoc = new ActiveXObject('Microsoft.XMLDOM');
10            xmlDoc.async = 'false';
11            xmlDoc.load(url);
12            return xmlDoc;
13        }
14    }
```

Obrázek 13: Metoda pro načtení XML dokumentu

Můj `XmlParser` umožňuje parserování *XML* do *DOM* na základě jeho zdrojového kódu, nebo URL adresy. Protože k úloze převodu *XML* do *DOM* má blízko úloha zpětného převodu (serializace *XML*, neboli převod *DOM* do textového řetězce), přidal jsem do `XmlParseru` i metody na serializaci. `XmlParser` navíc dokáže vrátit zdrojový kód *XML* na základě URL. Na obr. 13 je součást `XmlParseru` načítající *XML* z jeho URL. Je to metoda, která pomocí argumentu `url` vrátí referenci na nově vytvořenou instanci třídy *XML DOM* dokument. K tomu využívá jednoduchý HTTP požadavek (součást specifikace *DOM level 2*), jehož výsledek vrátí. V případě *IE*, který *DOM level 2* neimplementuje, se použije proprietární API Microsoftu. Řádek 10 určuje, že skript nejprve počká na odpověď. To je důležité, protože jinak by načítání *XML* probíhalo paralelně na pozadí a před jeho dokončením by výsledek obsahoval nulovou referenci. Ekvivalentem u HTTP požadavku je parametr `false` v metodě `open` na řádku 4.

Při serializaci je použit obdobný postup. Opět je to *DOM level 2*, kde je definována třída `XMLSerializer` a API Microsoftu. V ní je definována vlastnost `xml`, která slouží pouze pro čtení a ukládá text představující příslušnou *XML* větev (viz obr. 14).

Ačkoliv takto optimalizovaný `XmlParser` funguje ve všech webových prohlížečích, není výsledek všude stejný. Je to proto, že specifikace *DOM level 2* se liší od implementací Microsoft

```
1       serialize = function(dom)   {
2           if (dom.xml) return dom.xml;
3           var serializer = new XMLSerializer();
4           return serializer.serializeToString(dom);
5       }
```

Obrázek 14: Serializace XML větve

API. Také to je ukázka toho, kdy přestože je JS kód standardně optimalizován a funkční, ve skutečnosti v různých prostředích vyprodukuje různý výsledek. To může v některých případech vyvolat nežádoucí chování výsledného skriptu. Takových případů je v JS mnoho a přináší to další komplikace během jeho optimalizace. *DOM parser* Microsoftu nezařazuje do výsledku textové větve obsahující pouze bíle znaky (tzv. *white spaces*, čili tabelátory a zalomení řádků), zatímco ostatní prohlížeče to dělají. Který z přístupů je lepší, je sporné. Po odstranění těchto větví je manipulace s *DOM* stromem rychlejší, nakonec tyto větve nenesou žádnou informaci tak proč je prostě nesmazat? Z hlediska *XML* sice doopravdy nenesou žádnou informaci, ale nesou informaci o tzv. „štábní kultuře“. Pokud chceme takto „osekaný“ *DOM* zobrazit v *non-WYSIWYG* režimu, chybí v něm původní „štábní kultura“ a výsledek vypadá dost nepřehledně. Další problém může nastat při procházení *DOM* stromem, protože elementy v *DOM* se „štábní kulturou“ mohou tvořit jinou větev stromu než v *DOM* zkonstruovaném na základě stejného dokumentu v IE.

3.2.2 Validace XML

Jednou ze silných stránek jazyka *XML* je jeho flexibilita. Každý uživatel si může definovat vlastní značky (*tagy*) a v jakém uspořádání je lze přidávat do *XML* dokumentů. Z toho plyne potřeba nově vzniklý jazyk založený na *XML* nějakým způsobem definovat. Definicí *XML* jazyka je tzv. *schéma* a existuje několik formátů, jsou to *DTD* které mimo jiné používá jazyk *HTML* (pochází z dob společného předka *HTML* a *XML* — jazyka *SGML*), modernější *XML Schema* a *RELAX NG*.

Validace je v technologii *XML* proces, během kterého je analyzován cílový *XML* dokument, za účelem kontroly správnosti konzistence dat vzhledem k danému schématu. Je to důležitý proces, protože nevalidní *XML* by mohlo způsobit problémy během dalšího zpracování. Krom obecných pravidel určujících jak má korektní *XML* dokument vypadat (např. *tagy* musí být malými písmeny), je důležité se během psaní *XML* dokumentů držet daného schématu. Tzn. přidávat do dokumentu pouze elementy (resp. atributy), které jsou definovány ve *schématu*. *Schéma* navíc vymezuje kam je možné daný element přidat.

Kontrolu validity provádí *parser* (viz kapitola 3.2.1). *Parsery* mívají vlastnosti běžných konverzačních (interaktivních) překladačů, takže se dají integrovat do nástrojů pro práci s *XML*. Dokáží uživatele upozornit na řadu běžných chyb, včetně jejich pozice. V případě webových *parserů* je při výskytu chyby výstupem nulová reference a ne reference na nově vytvořený *DOM*. Platí to i v případě, že *XML* data neodpovídají zadanému *schématu*. Jednou z dobrých vlastností *XML* editorů je, že pomocí *XML parseru* dokáží nabídnout uživateli takové operace, které jsou v daném kontextu platné. To je velmi užitečné, protože se tak dá předejít mnoha chybám.

O kontrolu validity se v případě mého editoru stará objekt `xmlValid`. Jeho hlavní úlohou je ověřování operací typu vkládání a mazání elementů. Kdykoliv je potřeba, lze položit `xmlValidu` jednoduchý dotaz formou jeho metody, jejíž návratovou hodnotou je hodnota typu *boolean*. Mezi tyto metody patří: `canAddElement()`, `canRemoveElement()` a `canChangeAttribute()`, neboli: mohu přidat element, mohu odebrat element, mohu změnit atribut. Funguje tak, že nejprve provede požadovanou operaci (např. přidá do dokumentu nový element) a pomocí `xmlParseru` určí, zda je výsledek validní. Poté vrátí dokument do původní podoby a návratové hodnotě předá výsledek testu.

Nejproblematičtější byla implementace metody `canRemoveElement()`. Nejprve jsem zkoušel pro ověření možnosti odebrání elementu cílovou větev naklonovat, poté odebrat z *DOM* stromu a po testu validity zpětně připojit klon. Bohužel tento způsob má drobný nedostatek, který způsoboval nesprávnou činnost jiného skriptu. Naklonovaná *XML* větev má jinou adresu v paměti než původní a proto je vedlejším efektem této metody jiná pozice testované větve v paměti počítače, protože je tvořena svým klonem. Problém nastane v případě, kdy před průběhem takto implementované metody odkazuje na testovanou větev nějaká reference. Tato reference se tak stane neplatnou (začne odkazovat na `null` čily žádný objekt). Uvědomil jsem si, že abych se vyhnul potřebě klonování za účelem zachování mazaného objektu, budu potřebovat metodu, která na místo smazání *DOM* větve jí pouze odpojí, takže bude nadále fyzicky existovat, jenom už nebude součástí původního stromu *DOM*. Po skončení kontroly validity je větev zpětně připojena na původní místo a výsledkem je původní *DOM*. Hledal jsem ve specifikaci *DOM level 1* a metodu pro odpojení větve jsem nenašel, naštěstí jsem ale našel v dokumentaci *API jQuery* metodu `detach()` vyhovující potřebám této rutiny.

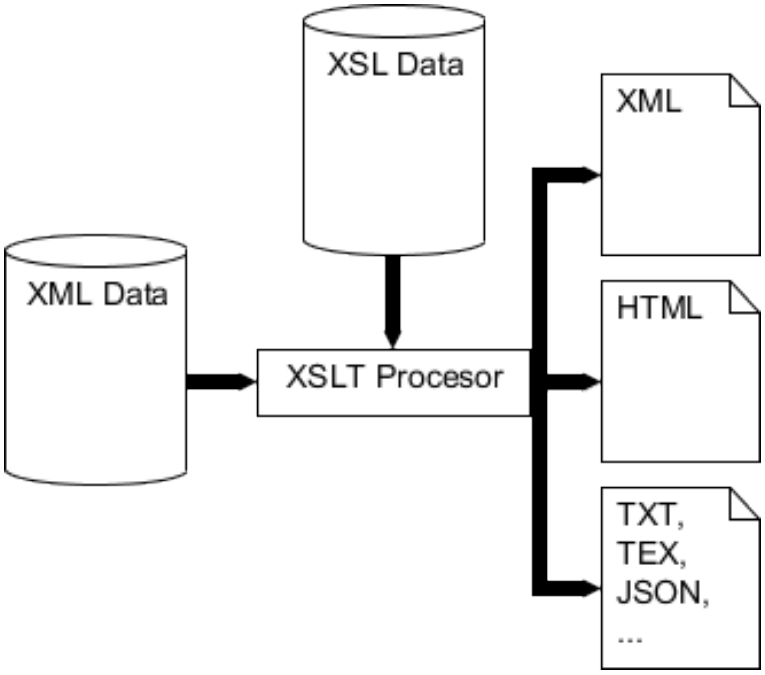
Uživatel může používat můj editor i v případě, že v XML dokumentu chybí informace ohledně použitého schématu. V takovém případě platí volná pravidla a lze přidávat a odebírat libovolné elementy (kořenový element nelze odebrat nikdy).

3.3 Transformace XSLT

XML samo o sobě vyjadřuje sémantiku dat, které popisuje. Všechny *XML* schémata by se měla řídit tímto pravidlem a nedefinovat např. elementy informující o formátování resp. sazbě. To je jeden ze zásadních rozdílů oproti jazyku *HTML*. Stejně jako *HTML* dokumenty i *XML* dokumenty lze formátovat pomocí kaskádových stylů (*CSS*). Dokonce lze ve webovém prohlížeči zobrazovat *XML* dokumenty formátované pomocí *CSS*.

3.3.1 XSLT a důvod jeho použití

Výhodou jazyka *XML* oproti *HTML* je podpora transformací *XSLT*. Ta pracuje s *XSL* styly, což je stylový jazyk navržený přímo pro *XML*. Během vývoje této práce jsem zvažoval, který z jazyků pro formátování *XML* využiji pro umožnění *WYSWYG* režimu. V něm je nezbytné *XML* data nějakým způsobem formátovat. Jaké jsou tedy pro a proti jazyka *XSLT* oproti *CSS*?



Obrázek 15: Princip XSLT (zdroj: vlastní)

Při volbě stylového jazyka pro *XML* platí jednoduché, ale trefné rčení: „použij *CSS* když můžeš a *XSL*, když musíš“. *CSS* je jednoduché na naučení i na použití, proto mnoho *WYSWYG* editorů pro *XML* využívá *CSS*. Jednoduchost *CSS* je však zároveň jeho slabou stránkou. Za cenu určité složitosti dostane uživatel volící formátování pomocí *XSL* stylů širší možnosti a flexibilitu oproti

CSS. Vývoj *XSL* stylů tak trochu připomíná programování, existují v něm smyčky a rozhodovací konstrukce obdobně jako v strukturovaném programování. Soubor obsahující *XSL* styl jsou vstupní data pro transformaci *XSLT* (princip je znázorněn na obr. 15), na základě kterých se generuje výstup. Ten může představovat prakticky jakýkoliv textový formát, např. jiné *XML*, často to bývá *HTML*. Uspořádání výstupu *XSLT* transformace se může zásadním způsobem lišit od původního uspořádáni ve zdrojovém *XML*. Možnost libovolného výstupního formátu a uspořádání dělá z *XSLT* široce použitelný nástroj. Díky možnostem, které nabízí *XSLT* transformace jsem se rozhodl implementovat jej do vyvíjeného *XML* editoru. Dalším důvodem je fakt, že většina uživatelů volí jazyka *XML* právě díky možnosti provádět tyto transformace.

3.3.2 Implementace XSLT procesoru

Transformace *XSLT* má na starost *XSLT procesor* (viz obr. 15). Existuje řada volně dostupných *XSLT procesorů* prakticky pro všechny platformy. Časem začali implementovat *XSLT procesor* i webové prohlížeče, první z nich byl IE verze 5.0, následoval Firefox verze 3 a Opera verze 9.

Stejně jako v případě `XmlParseru` (viz kapitola 3.2.1) způsob použití *XSLT procesoru* ve webovém prohlížeči se liší v IE. Abych tyto rozdíly sjednotil, definoval jsem třídu `XsltProc`, která na základě dostupných prostředků provádí transformace.

```
1     transformFromUrl = function(xmlParser,xml,xslUrl) {
2         var xsl = xmlParser.urlToDom(xslUrl);
3         if (window.ActiveXObject) {
4             res = xml.transformNode(xsl);
5         }
6         else if (document.implementation && document.implementation.createDocument) {
7             xsltProcessor = new XSLTProcessor();
8             xsltProcessor.importStylesheet(xsl);
9             resultDocument = xsltProcessor.transformToFragment(xml,document);
10            res = xmlParser.serialize(resultDocument);
11        }
12    }
```

Obrázek 16: Metoda pro transformaci XSLT

Třída `XsltProc` obsahuje pouze jednu metodu, pro vykonání samotné transformace. V jejím argumentu jsou předány: reference na instanci `XmlParseru`, XML zadaný jako DOM, který se má transformovat a URL adresa *XSL* stylu. Vše je ukázáno na obr. 16. Stejně jako při serializaci v IE je *XSLT* otázkou jednoho řádku kódu (viz 4. řádek). Větve *DOM* v IE totiž obsahují metodu `transformNode()`, to je jedna z mnoha věcí v rozporu se standardy W3C. V ostatních případech je vytvořena instance `xsltProcesor`, která představuje interní *XSLT procesor* prohlížeče, které je předán *XSL* styl. Výsledkem je tzv. *dokument fragment*, co je to

dokument fragment není v této rutině důležité, důležité je, že je kompatibilní s instancí *DOM node*. Tzn., že jí objekt předaný v referenci `xmlParser` umí serializovat, takže je vrácen textový řetězec představující výsledek transformace. Serializace je důležitá, aby byl výsledek schodný s výsledkem metody `transformNode()` implementované v IE.

3.3.3 Možnosti výsledného modulu pro XSLT

Třídy `XmlParser`, `XmlValid` a `XsltProc` jsou základní prvky, které používá můj editor. Nevýhodou `XsltProc` je, že nevrátí správný výsledek, pokud prohlížeč nepodporuje *XSLT*. Na druhou stranu, pomocí `XsltProc` je možné transformovat libovolnou *XML* větev, takže není potřeba transformovat pokaždé celý dokument. Tuto vlastnost *XSLT* nevyužívá pouze můj editor, jde o obecně známou alternativu *Ajaxu* (jazyk pro asynchronní zpracování *XML* v JS).

3.4 Zvýrazňovač syntaxe

S rozvojem programování roste potřeba dobré orientace v různých zdrojových kódech. Kódy mohou být obtížně čitelné i přesto, že jejich kodér dodržuje vhodnou štábní kulturu (organizace tabelátorů a zalomení řádek). Proto každý kvalitní editor zdrojového kódu formátuje text tak, aby v něm vyznačil například klíčová slova, komentáře, či názvy identifikátorů. Kodérovi tak usnadní práci a někdy pomůže odhalit chybu dříve než kompilátor nebo interpret. Program resp. objekt, který má za úkol takto formátovat zdrojové kódy nazvěme zvýrazňovač syntaxe (angl. *syntax highlighter*, dále ZS).

Vývoj takového ZS není triviální záležitostí a v jazyce JS náročnost ještě roste kvůli potřebě optimalizace. Možná proto webové editory zvýrazňování syntaxe nepodporují. Ve webové úschovně pluginů napsaných v *JQuery* se mi podařilo najít několik zvýrazňovačů syntaxe, bohužel všechny slouží pouze ke generování statického obsahu. Jsou tedy určeny spíše vývojářům webu, kteří chtějí na své stránky umístit ukázky zdrojových kódů, než vývojářům webových editorů. Protože bych možnost zvýrazňování syntaxe rád implementoval do svého editoru, rozhodl jsem se, že se pokusím napsat vlastní ZS, který bude pracovat i během editačního režimu.

3.4.1 Návrh

Pro vývoj mého ZS se hodí použít komponentu *RE* (viz kap.1), kterou jsem napsal v rámci svého magisterského ročníkového projektu. Můj ZS ji použije pro *editační režim* a abych zajistil

```
1       /* returns minimum of a and b given by argument */
2       function min(a,b) {
3           return (a < b) ? a : b;
4       }
```

Obrázek 17: Funkce vracejici minimum dvou čísel v JavaScriptu

lepší použitelnost svého pluginu, umožním i *generování statického obsahu*. Také jsem se rozhodl použít pro ukázky zdrojových kódů v této práci právě svůj ZS.

Pro každý jazyk existuje nespočet možností, co a jak by měl označovat. Je tedy důležité ZS do značné míry *ladit* — zkoušet různé možnosti a porovnávat výsledky. Aby byl ZS dobře odladitelný, je důležité oddělit od sebe problematiku pravidel na základě kterých pracuje a problematiku samotného algoritmu, který se postará o formátování. Nejdříve jsem tedy sestavil rozhraní pro pravidla ZS a pak jsem se postaral o to, aby na základě kolekce těchto pravidel ZS pracoval. Takto napsanému ZS se mohou předat různá pravidla a realizovat tak ZS různých jazyků. Volil jsem i možnost tato pravidla načítat z dat ve formátu *XML*. Stačí tedy jeden skript, a aniž by se měnil jeho zdrojový kód, může se pomocí něho realizovat ZS pro jakýkoliv jazyk.

Vzhled formátovaných částí textu u webového ZS je určen pomocí *CSS* definic. ZS tedy změní libovolné *HTML* na *HTML* obsahující stejný text tak, že jsou v něm jednotlivé části rozděleny do elementů *span* s příslušným atributem *class*. Uživatel tak může připojením různých souborů s *CSS* definicemi ZS snadno tzv. „*skinovat*“. Takto navržený webový ZS využívá standardní formáty (*XML* a *CSS*), které jsou navíc realizovány textovými soubory. Umožňuje tak pohodlný vývoj různých nástrojů pro tento plugin (např. i v jiných jazycích než v JS).

Abych přiblížil, jak ZS pracuje, uvádím zde příklad na obrázku 17 týkající se jazyka JS. Je na něm jednoduchá funkce vracející minimum dvou čísel definovaných argumentem funkce (identifikátory *a* a *b*). Ukázka obsahuje následující klíčová slova: `function` a `return`, které je potřeba zvýraznit. Stejně tak je vhodné zvýraznit i *symboly závorek*, *otazník*, *dvojtečku*, *znaménko menší než*, *čárku* a *středník*. Dále je zde slovo `min`, což je identifikátor funkce a v poslední řadě *komentář*, kterým ukázka začíná.

Nechť je dána forma pravidel podle kterých ZS označuje nějaký zdrojový kód. Uspořádanou k-tici instancí těchto pravidel nazvu *kolekcí pravidel* ZS. Záleží zde na pořadí pravidel v kolekci, neboť různá pravidla mají různou prioritu. Pokud např. ZS označuje v JS text uvnitř komentáře, klíčová slova a ostatní symboly v něm nehrají roli.

ZS musí rozeznat, které úseky textů má zvýrazňovat. Pravidla tedy obsahují kromě názvu třídy pro *CSS* styl i *startovní* a *koncový řetězec*. Např. pro komentář v ukázce je startovním

řetězcem „/*“ a koncovým „*/“. Abych vymezil klíčová slova, použiji pro ně jako startovní řetězec ono klíčové slovo a prázdný řetězec jako koncový. Symboly *závorek*, *otazník*, *dvojtečka* atd. se dají chápat jako klíčová slova délky jedna.

U startovních a koncových řetězců je navíc potřeba rozeznávat, kdy se jedná o *slovo* a kdy o *prostý řetězec*. Slovem je myšlen řetězec oddělený od zbytku textu *bílými znaky*. Např. v případě, že by se v ukázkovém zdrojovém kódu nacházelo `returns` namísto `return` (mohlo by jít o překlep), nechci aby v něm bylo `return` zvýrazněno a je to třeba vymezit v definici pravidla. Podle mého návrhu tedy řetězec končící mezerou bude slovo nikoli prostý řetězec.

3.4.2 Implementace zvýrazňovače syntaxe

Můj ZS pracuje tak, že analyzuje daný text řádek po řádku a výsledek zapisuje na výstup. Stěžejní je tedy metoda, která na základě vstupního řetězce vrátí řetězec představující *HTML* kód s příslušnými elementy *span*. Pokud ZS během analýzy textu narazí na startovní řetězec některého pravidla z kolekce s vyšší prioritou, než je pravidlo právě aktivní, přestane být toto pravidlo aktivním namísto pravidla právě nalezeného. ZS navíc musí počítat s více pravidly najednou, např. v ukázce na obr. 17 je klíčové slovo `function`, které je označeno a navíc je `function` startovním řetězcem pro vyznačení identifikátoru funkce `min`. Proto každé další aktivní pravidlo ukládám na vrchol FILO zásobníku (první dovnitř, poslední ven), a po jeho skončení ho z něj opět odeberu, takže na jeho vrcholu zůstane pravidlo předchozí. Tento proces funguje bezproblémově a na základě různých kolekcí pravidel ZS produkuje různé výsledky, přesně podle očekávání.

```
1       <body>
2               <!--varianta a)-->
3               <p>řádek jedna</p>
4               <p>řádek dvě</p>
5               <p>řádek tři</p>
6               <!--varianta b)-->
7               <p>řádek jedna</p>
8               řádek dvě<br />
9               řádek tři
10              <!--varianta c)-->
11              <p>řádek jedna<p>řádek dvě</p>řádek tři</p>
12      </body>
```

Obrázek 18: Možnosti řádkování v HTML

Když je vyřešen problém samotného zvýrazňování textu, není problém použít takovýto ZS pro generování statického obsahu. Zbývá přizpůsobit ZS tak, aby umožňoval i editační režim. V tomto

případě musí ZS reagovat na každou uživatelskou změnu obsahu dokumentu přeformátováním dokumentu tak, aby vše sedělo. Přeformátováním dokumentu myslím změnu jeho *HTML* kódu na základě kolekce pravidel ZS. Je zřejmé, že při změně *HTML* dokumentu skriptem dojde k vyresetování navigačních prvků editačního režimu. Jde o nastavení pozice *caret* a posuvníků (*scroll offsets*) na nulu. Skript se musí postarat o jejich nastavení na příslušnou pozici tak, jak tomu bylo ještě před přeformátováním.

Protože metoda, pomocí níž ZS pracuje, má na vstupu řetězec představující konkrétní řádek, musí se celý text této metodě předat řádek po řádku. V případě generování statického obsahu zde není problém, řádky předaného textu jsou v něm jednoduše odděleny pomocí tzv. *escape sequence* „n“[3]. Ve chvíli, kdy jde o editační režim je situace horší. V jazyce *HTML* totiž existuje několik způsobů jak zakončit řádek. Na obrázku 18 uvádím tři příklady, které významově všechny představují tři řádky v jazyce *HTML*, avšak všechny jsou představovány jiným *DOM* stromem.

Formálně nejlepší je bezesporu varianta *a)*. Ve variantě *b)* je dokonce text mimo odstavec (takže tvoří přímo součást elementu `body`), což není příliš vhodné a osobně považuji přítomnost elementu `br` za nešťastnou. Varianta *c)* na tom není formálně o nic lépe, nicméně i tento kód prohlížeče akceptují. O zdrojové *HTML* se u *WYSIWYG* editorů stará samotný *engine* editoru (v případě JS prohlížeč) jejich velkou slabinou je právě fakt, že výsledné zdrojové kódy produkované *WYSIWYG* režimem na tom nejsou formálně nejlépe.

Provedl jsem řadu testů, při nichž jsem zjišťoval, jak různé prohlížeče drží formu naznačenou na obrázku 18 u varianty *a)* i v případě, že uživatel provedl s dokumentem řadu nejrůznějších změn. Zejména jde o operace, které nějakým způsobem přidávají (případně mažou) řádky. Výsledkem těchto testů je fakt, že IE snad ve všech případech drží správnou formu odstavců, nehledě na to, co se s dokumentem děje. Naproti tomu ostatní prohlížeče si vedou přímo tragicky. V případě vkládání nových řádků vloží do dokumentu jednou `br`, jindy zase `p`, podle toho jak se jim právě umane. Jakmile začne uživatel pracovat se schránkou, tak dokonce vyprodukují vnořené odstavce tak, jak je tomu ukázáno ve variantě *c)*. Navíc se několikrát stalo, že se text, který byl původně v elementu `p`, dostal do elementu `body` (s tím, že byl původní element `p` smazán) obdobně jako je ve variantě *b)*. Aby byl můj editor kompatibilní s různými prohlížeči, je tedy třeba brát v potaz všechny výše zmíněné případy. Vyvinul jsem algoritmus, který na základě různé *DOM* struktury dokumentu vrátí příslušné řádky tak, že počítá se všemi možnými variantami.

[3] V IE escape sequence nefungovala, ale řádky v textu se dají rozlišit i podle znaku s *ASCII* kódem 13 (to však nefunguje v ostatních prohlížečích, v Opeře fungují obě možnosti).

3.4.3 Výsledný plugin prakticky

V této kapitole ukážu, jak použít můj ZS jak pro generování statického obsahu, tak pro vývoj editoru zdrojového kódu v podstatě jakéhokoliv jazyka. Zároveň je i výzvou pro ostatní vývojáře, kteří by chtěli přispět k rozvoji tohoto pluginu, např. napsáním dalších pravidel pro nejrůznější jazyky.

```
1       var rc = new $().rc();
2       rc.loadFromXML("rules.xml");
3       $.fn.sh().setRuleCol(rc);
4       // only for richEdit mode
5       $(window).load( function() {
6               $.fn.sh().init();
7       } );
8       // ~
```

Obrázek 19: Inicializace zvýrazňovače syntaxe

Inicializační kód ZS by mohl vypadat např. tak, jako v ukázce na obrázku 19. První řádek je alokování samotného objektu představující kolekci pravidel. Jednotlivá pravidla je možno načíst pomocí API tohoto objektu, alternativně lze použít metodu `loadFromXML()`, které je předána cesta k *XML* souboru, ve kterém jsou jednotlivá pravidla popsána. Následně je předána reference na kolekci pravidel samotnému ZS, ten v tuto chvíli ví, podle kterých pravidel má pracovat. Touto metodou lze za běhu libovolně volit mezi různými kolekcemi pravidel, stačí předat příslušnou referenci. Je to užitečné zejména v případě, kdy chcete např. na jedné webové stránce mít zvýrazněno několik ukázek zdrojových kódů různých jazyků. Následující řádky jsou potřebné pouze v případě, že používáme plugin jako editor.

Protože by bylo zbytečně zdlouhavé probírat zde všechna pravidla pro zvýrazňování JS, uvedu pouze ty potřebné pro zvýraznění kódu v předchozí ukázce na obr 17. Jejich *XML* kód podle konvence, kterou jsem pro tyto účely navrhl, je uveden v ukázce na obr. 20. Textový obsah elementů *rule* představuje název třídy náležející příslušnému pravidlu. Jeho atributy *start* a *end* představují startovní a koncové řetězce. Důležité je, že komentář má nejvyšší prioritu, proto musí být uveden jako první. Dále se nesmí zapomenout uvést mezera na konci startovního řetězce `function` a `return`, aby byl řetězec brán jako slovo zakončené bílým znakem. Zbývá dodat *CSS* definice tříd `comment`, `sym` a `keyword`.

Poslední záležitost, týkající se praktického využití mého ZS, je formátování statického obsahu, např. formátování ukázek kódu na webové stránce. Ty bývá zvykem na stránky vkládat do elementu `pre`, kvůli zachování oddělovacích symbolů tak, jak jsou ve zdrojovém kódu. Dejme

```
1     <?xml version="1.0"?>
2     <rules>
3           <rule start="/*" end="*/">comment</rule>
4           <rule start="(" end="">sym</rule>
5           <rule start=")" end="">sym</rule>
6           <rule start="{" end="">sym</rule>
7           <rule start="}" end="">sym</rule>
8           <rule start="," end="">sym</rule>
9           <rule start="<" end="">sym</rule>
10          <rule start="?" end="">sym</rule>
11          <rule start=":" end="">sym</rule>
12          <rule start=";" end="">sym</rule>
13          <rule start="function " end="">keyWord</rule>
14          <rule start="return " end="">keyWord</rule>
15          <rule start="function " end="(">functionName</rule>
16    </rules>
```

Obrázek 20: Pravidla pro zvýrazňovač ve formátu XML

tomu, že máte na svých stránkách několik elementů `pre` s ukázkami kódů a právě jste si stáhli z webu můj plugin. Provedli jste potřebné inicializace pluginu a máte *XML* i *CSS* pro ZS odpovídající vašim potřebám. Stačí vložit skript, který pomocí *CSS* selektoru předá elementy `pre` *jQuery* a zavolat metodu `formatStatic()` a můžete plugin používat.

3.4.4 Klady a zápory zvýrazňovače

Plugin pro zvýrazňování syntaxe se ukázal velmi užitečným pro dokumentaci tohoto projektu. Dá se využít pro jakýkoliv jazyk a umožňuje i editační režim. Navíc ukázal i užitečnost jazyka *XML*, který je použit pro popis instancí objektů, což je jeho vlastnost používaná např. v databázových aplikacích. Protože se pohybuji na poli JS, dal by se místo něj použít jazyk *JSON* (JavaScript Object Notification), protože však vyvíjím editor pro *XML*, volím raději technologii *XML*. Jeho nevýhodou je poměrně velký objem, při řešení výše zmíněných problémů skript „*vyrostl*“.

3.5 WYSIWYG režim

WYSIWYG režim je mód, ve kterém uživatel pracuje s osazeným dokumentem. To je z uživatelského hlediska pohodlné, může tak např. vytvářet webové stránky bez jakékoliv znalosti jazyka *HTML*. Na druhou stranu je zde určité omezení, protože uživatel může s dokumentem provádět pouze takové operace, které mu nabízí uživatelské rozhraní. Webové *WYSIWYG* editory bývají kritizovány kvůli faktu, že jejich výsledkem často bývá nešikovné, nebo nevalidní uspořádání elementů. Na druhou stranu výhody *WYSIWYG* režimu jsou důležité zejména při vývoji webových

editorů.

Problematika *WYSIWYG* režimu úzce souvisí s návrhem uživatelského rozhraní (viz kapitola 3.6). Nejprve je ale důležité vymyslet způsob jakým se realizuje samotný *WYSWYG* režim pro dokumenty formátu *XML*. Protože samotné *XML* neobsahuje informace o jeho formátování, je zásadní otázkou *WYSIWYG* režimu pro *XML* výběr stylového jazyka. V kapitole 3.3.1 jsem uvedl proč jsem zvolil stylový jazyk *XSL*, čili možnost transformovat *XML* pomocí *XSLT*. Výstup *XSLT* transformace je možné dál zpracovávat pomocí formátovacích objektů. Protože tato úloha není jednoduchá, jde o poměrně problematickou pasáž procesu *XSLT* a lze takto formátovat *XML* pouze použitím externí aplikace. Z bezpečnostních důvodů nelze v JS využívat externí aplikace. Protože JS běží na straně klienta, otevřelo by to cestu nežádoucímu software.

Další možností formátování *XML* pomocí *XSLT* je generovat *HTML* výstup. *HTML* dokumenty jsou osazeny pomocí *CSS* a ty je možné vkládat přímo do zdrojového kódu dokumentu. Možnost generování *HTML* přímo ze zdrojového *XML* patří mezi oblíbenou vlastnost transformace *XSLT* a protože vyvíjím webový editor, realizoval jsem pomocí ní *WYSIWYG* režim vyvíjeného *XML* editoru.

3.5.1 Nejednoznačnost XSLT

Formátování pomocí *CSS* spočívá k přiřazení určitých vlastností vypovídajících o vizuálním charakteru jednotlivým elementů. Díky tomu je možné zobrazit *XML* přímo tak jak je bez toho, aniž by původní dokument jakkoliv měnil svojí strukturu. Tato vlastnost přináší určitou jednoduchost při vývoji *WYSIWYG* režimu za použití *CSS* stylů a zřejmě i proto existuje řada webových *WYSIWYG* editorů založených na *CSS*, ať už jde o editory pro jazyky *XML* nebo *HTML*.

V případě *XSLT* jako nástroje pro *WYSIWYG* režim je to problematičtější. Při transformaci *XML* do *HTML* podoby pomocí *XSLT* se ztrácí zpětná vazba na původní *XML*. Na jedné straně existuje *XML* dokument, se kterým pracuje editor, na druhé straně je *HTML*, představující jeho vizuální podobu. Ačkoliv je tento *HTML* dokument vygenerován automaticky procesorem *XSLT*, nejsou tyto dva dokumenty navzájem nijak provázány.

Nabízí se jednoduché řešení generovat původní *XML* na základě výsledného *HTML* a *XSL* předpisu, ve chvíli kdy je potřeba pracovat s daty *XML*. Protože je *XSLT* transformace nejednoznačná, nalezení algoritmu, který by jednoznačně řešil tuto úlohu je problém. Projevem nejednoznačnosti je, že pro všechny *XML* elementy nemusí existovat transformační pravidla v šablonách *XSL*, pomocí kterých se transformace provádí. Z principu na kterém jsou založeny

transformace *XSLT* je zřejmé, že na základě výstupního *HTML* a *XSL* stylu se dá určit prakticky nekonečně mnoho různých *XML* na základě kterých transformace proběhla.

I při nalezení algoritmu generující *XML* na základě výsledku transformace a stylu *XSL* by existovala omezení a algoritmus by nikdy nemohl fungovat obecně, ale pouze pro omezenou podmnožinu *XSL* stylů. Přesto by nalezení takového algoritmu nebylo jednoduchou úlohou.

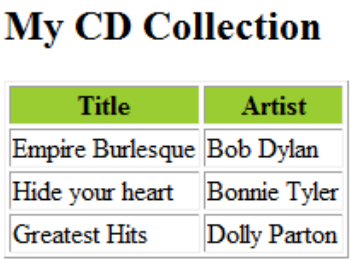
Rozhodl jsem se proto hledat jiné řešení. *XSLT* je natolik obecný a flexibilní proces, že existuje řada různých možností jak psát *XSL* a generovat jeden a ten samý výstup. U *WYSIWYG* editorů je důležité aktualizovat zobrazené elementy po částech, protože „refresh“ celého dokumentu (pokaždé když se změní některá jeho část) by výrazně zpomalovala běh editoru. Jak jsem naznačil v kapitole 3.3.3, lze pomocí *XSLT* transformovat pouze určitou část. Na základě této vlastnosti *XSLT* mně napadla možnost, že by se během transformace jednotlivých částí ukládal do *HTML* výstupu odkaz na původní *HTML* element. Jednou z funkcí *jQuery* je totiž možnost přiřazovat *HTML* elementům libovolná data (v případě *XML* elementů to nelze). Skript se postará o to, aby se výstup transformovaného *XML* elementu obalil do elementu `span`, kterému se přiřadí data obsahující referenci na původní *XML* větev. Při práci s libovolným *HTML* elementem ve *WYSIWYG* režimu se pak dá jednoduše odkázat na otcovský element `span` a následně získat reference na větev stromu *XML*.

3.5.2 Řešení WYSIWYG režimu

Styly *XSL* jsou psány ve formátu *XML*. Pravidla pro transformaci *XSLT* jsou v nich zadány pomocí tzv. šablon (angl. *templates*). Ty obsahují hlavičku, která určuje podmínku, kdy se šablona aplikuje a v jejím těle jsou příkazy pro zpracování. Příkaz `apply-templates` např. *XSLT* procesoru říká, že má pokračovat ve zpracování a aplikovat tak další šablony pro vnořené elementy, které tvoří obsah elementu který vyhověl šabloně obsahující tento příkaz. Pokud *XSLT* procesor narazí na text, který netvoří žádnou instrukci *XSL*, je tento text přímo zapsán na výstup.

Při testování *XSLT* v JS jsem využil jednoduchý příklad, který je dostupný na výukových stránkách [2]. Jde o jednoduchý katalog cd, který se zformátuje jako *HTML* tabulka. Kořenový element je `catalog`, který se převede jako *HTML* nadpis a element představující tabulku. Také obsahuje elementy `cd`, tvořené jednotlivými řádky této tabulky. Každé `cd` má pět položek, čily dceřiných elementů. Ve výsledné *HTML* tabulce se však zobrazují pouze položky `artist` a `title`.

Existuje několik způsobů, jak napsat *XSL* předpis, na základě kterého se vygeneruje cílová tabulka (viz kapitola 3.5.1). Styl pro cd katalog dostupný na stránkách [2] je tvořen pouze



Obrázek 21: Zformátovaný XML dokument

jednou šablonou, která se aplikuje na kořenový element. Tento způsob neumožňuje transformovat samostatně větve cd katalogu. Např. pokud se předá *XSLT* procesoru element `cd`, nenajde se k němu žádná šablona.

Jiný způsob, jak psát *XSL* šablony je definovat pro každý element, který se má formátovat, jeho vlastní šablonu. Díky tomu je možné formátovat libovolně kteroukoliv větev *XML* nezávisle na zbytku stromu a to otvírá cestu řešení *WYSIWYG* režimu které jsem naznačil v kapitole 3.5.1. Z toho plyne určité omezení, uživatel vyvíjející vlastní styl pro zobrazování *XML* ve *WYSIWYG* režimu mého editoru musí styl napsat výše zmíněným způsobem. Pro případ cd katalogu je příslušný styl, se kterým dokáže pracovat můj editor, ukázán na obr. 22. Atribut `contentediable` říká, že uživatel může měnit obsah elementu. Výsledný dokument, tak jak se zobrazí ve *WYSIWYG* režimu, je uveden na obr. 21.

3.5.3 Implementace WYSIWYG režimu

O *WYSIWYG* režim se v případě mého editoru stará instance třídy `Wysiwyg` (viz kapitola 3.1). Protože poskytuje uživateli možnost editovat data formátu *HTML*, je odvozena formou dědičnosti od třídy `richEdit`. Třída *WYSIWYG* svému okolí poskytuje dvě metody: `refreshAll()` a `refreshNode()`. Ta první provede transformaci celého *XML* dokumentu a je volána během inicializace editoru. Druhá provede aktualizaci cílového elementu, tzn. pomocí *XSLT* převede sebe a všechny své dceřiné větve na *HTML* a nahradí obsah původního *HTML*.

Obě metody `Wysiwygu` volají soukromou metodu `transformNode()`, která vrátí transformaci větve (včetně dceřiných) předané jejím argumentem. Ta je stěžejní pro funkčnost *WYSIWYG* režimu a důležité je, že do výsledku transformace ukládá odkazy na původní *XML* elementy. Aby to bylo možné je potřeba zajistit, aby *XSLT* procesor nezpracovával rekurzivně další elementy při výskytu `apply-templates`, protože `transformNode()` provádí postupně

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="catalog">
    <h2> My CD Collection</h2>
    <table border=''>
        <tr bgcolor="#9acd32">
            <th>Title</th>
            <th>Artist</th>
        </tr>
        <xsl:apply-templates />
    </table>
</xsl:template>

<template match="catalog/cd">
    <tr>
        <xsl:apply-templates />
    </tr>
</xsl:template>

<template match="catalog/cd/artist">
    <td>
        <span contenteditable="true">
            <xsl:value-of select=".">
        </span>
    </td>
</xsl:template>

<template match="catalog/cd/title">
    <td>
        <span contenteditable="true">
            <xsl:value-of select=".">
        </span>
    </td>
</xsl:template>

<xsl:template match="catalog/cd/country"></xsl:template>
<xsl:template match="catalog/cd/company"></xsl:template>
<xsl:template match="catalog/cd/price"></xsl:template>
<xsl:template match="catalog/cd/year"></xsl:template>

</xsl:stylesheet>
```

Obrázek 22: Ukázkový styl pro umožnění WYSYWYG režimu

```javascript
_transformNode = function(doc,node,out) {
    /* transform */
    xslUrl = refXmlEdit.getXslUrl();
    var result = refXmlEdit.getXsltProc().transformFromUrl(refXmlEdit.getXmlParser(),node,xslUrl);
    result = result.replace('<?xml version="1.0"encoding="UTF-16"?>','');
    /* recursion */
    if (result != '') {
        /* create new node */
        var newNode = doc.createElement('span class="xmlRef"');
        $(newNode).html(result);
        /* save xml reference */
        $(newNode).data('xmlRef',node);
        /* ~ */
        ref = $(newNode).find('.apply-templates')[0];
        if ( $(out).attr('class') == 'apply-templates' ) {
            $(newNode).insertBefore(out);
        }
        if (ref) out = ref;
        $(node).children().each( function() {
            _transformNode(doc,this,out);
        } );
    }
    /* ~ */
}
```

Obrázek 23: Metoda transformNode

XSLT element po elementu a výsledky jednotlivých transformací postupně připojuje. Proto je metodě předán XSL styl, ve kterém jsou tagy `<xsl:apply-templates />` nahrazeny `<span class="apply-templates" />`. Pokud by tak nebylo, skript by nefungoval správně, protože by se o rekursivní zpracování vnořených elementů staral XSLT procesor namísto skriptu. Celý algoritmus je ukázán na obr. 23.

Nejprve je provedena transformace větve `node` předané argumentem pomocí instance třídy `xsltProc`. Instanci `xsltProc` uchovává objekt `xmlEdit`, a zpřístupňuje jí okolí pomocí metody `getXsltProc()`. Výsledek transformace obsahuje i hlavičku *XML* dokumentu, tu je potřeba odstranit (řádek 5). V případě, že výsledek transformace netvoří prázdný řetězec, skript pokračuje dál. Dále je vytvořen nový element `span`, který obalí výsledek transformace a přiřadí se mu reference na původní *XML* větev pomocí *jQuery* (řádek 12). Následně je vyhledán `span` třídy `apply-templates`, který upozorňuje skript na místo kam připojit další transformované větve podřízené momentální větvi. Nová větev uložená v proměnné `newNode` je připojena na místo předchozího výskytu elementu s atributem `class` o hodnotě `apply-templates` (řádek 16). Nakonec je uložena nová reference `out`, která tvoří nový cíl pro připojení podřízených větví a je provedeno rekurzivní volání metody, pro každou podřízenou větev.

3.5.4 Použití pluginu pro WYSIWYG režim

Přestože `Wysiwyg` nabízí pouze dvě metody, je velmi užitečný, protože se stará o tu nejtěžší práci spjatou s *WYSIWYG* režimem. Díky tomu můžou všechny všechny prvky editoru provádět operace na úrovni *XML* dokumentu a pomocí jedné metody udržovat funkčnost *WYSIWYG* režimu.

Pokud např. je potřeba změnit nějaký atribut, který má vliv na formátování, stačí nastavit jeho hodnotu pomocí JS a poté jednoduše zavolat metodu `refreshNode()` a jako argument předat referenci na element, jehož atribut se změnil. V případě, že je potřeba přidat nový element, stačí provést přidání elementu a zavolat metodu `refreshNode()` s argumentem otcovského elementu, do kterého byl nový element přidán. Při mazání je nutné nejprve najít *HTML* element, který obsahuje referenci na *XML* element, který se má odebrat, aby se před smazáním *XML* elementu smazal i *HTML* element tvořící jeho *HTML* podobu.

3.6 Uživatelské rozhraní

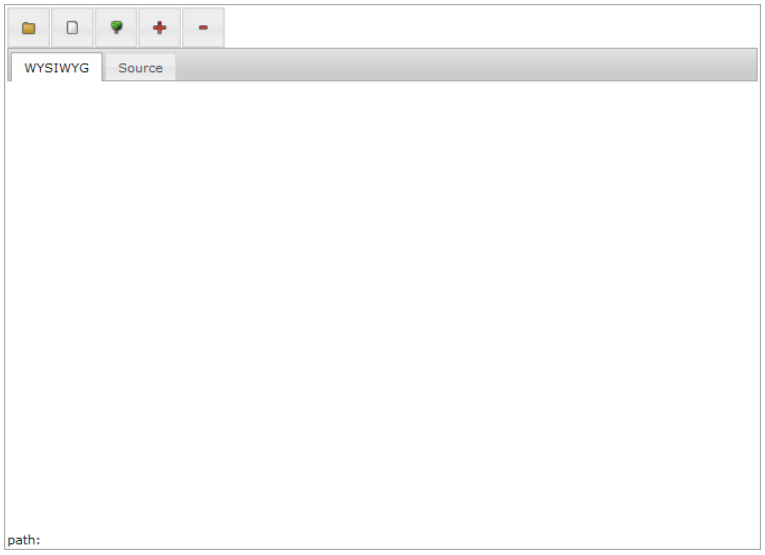
Uživatelské rozhraní (angl. zkr. UI) je prostředek, kterým komunikuje aplikace s uživatelem. V případě webových aplikací jde o grafické uživatelské rozhraní (zkr. GUI). GUI umožňuje

ovládat aplikaci pomocí interaktivních grafických prvků. Webové *WYSIWYG* editory bývají zpravidla ovládány formou jednoduchého ikonového menu, aby UI nezabral příliš místa na stránce.

Užitečnou funkcí frameworku *jQuery* je, že definuje standard pro vývoj *CSS* stylů definující *skiny* pro webové aplikace a přináší nástroje na jejich pohodlnou správu. *Skiny* definují vzhled UI a uživatel má možnost volit mezi sadou předdefinovaných *skinů* nebo vytvářet své vlastní.

3.6.1 Prvky uživatelské rozhraní

Rozhraní mého editoru se dá rozdělit na několik částí. Hlavní část je tvořena panelem, na kterém jsou umístěná ikonová tlačítka se základními funkcemi editoru. Pod ním je přepínač mezi *WYSIWYG* režimem a režimem zdrojového kódu, který se zobrazuje v hlavním okně editoru. V hlavním okně lze zobrazovat režim rozbalovacího stromu a úplně dole je *status*, ve kterém se vypisuje cesta právě editovaného elementu. Vše je vidět na obrázku 24.



Obrázek 24: Uživatelské rozhraní editoru

Framework *jQuery* nabízí efektivní prostředky pro implementaci dynamických (animovaných) ovládacích prvků na webové stránky. Dokonce lze pomocí *jQuery* vytvářet takové prvky UI, které nemají původ v jazyce *HTML*. Protože jde o poměrně robustní skripty, lze používat samotné jádro *jQuery* bez knihoven pro UI. Také lze spravovat *skiny* pohodlně prostřednictvím webové funkce *ThemeRoller* [3]. *Skiny* vytvořené pomocí *ThemeRolleru* jsou kompatibilní i s mojí aplikací.

3.6.2 Ovládací panel

Nejdůležitější prvek UI mého editoru je hlavní ovládací panel. Ten je tvořen několika tlačítky různé funkcionality. Dělí se na dvě části: první je statická a obsahuje obecné funkce pro práci s dokumentem, druhá se dynamicky aktualizuje a je spjatá s konkrétním *XML jmenným prostorem*. Mezi funkce ve statické části patří: načítání a ukládání dokumentu, přepínání režimu rozbalovacího stromu (viz kapitola 3.6.3) a obecné funkce pro přidávání a odebírání elementů.

Přidávání a odebírání elementů ve statické části funguje na základě dialogu, ve kterém uživatel vybere element, který se má do dokumentu přidat resp. odebrat a v případě přidávání také cíl kam se má nový element připojit. Aby aplikace věděla, které elementy uživateli nabídnout, je potřeba tyto elementy v inicializační části editoru zaregistrovat. To lze udělat jednoduše pomocí konfiguračního souboru editoru (viz 3.7.1). Nabídka cílových elementů, kam nový element připojit (a nabídka elementů k odebrání v případě mazání) je vygenerována na základě otcovských větví elementu, který obsahuje caret (kursor). Použitím instance třídy `xmlValid` je nabídka filtrována, aby obsahovala pouze validní operace. Díky tomu je zajištěno, že uživatel např. nemůže přidat nový element do kontextu ve kterém se nesmí vyskytovat.

O generování prvků ovládacího panelu se stará třída `UIhandler`. Ta generuje tlačítka pomocí jednoduché metody, ve které je předán název ikony tlačítka, *tooltip* (tj. nápověda zobrazená ve chvíli kdy se kursor myši nachází nad ovládacím prvkem) a *uzávěr* (neboli anonymní funkce) představující *handler* (tj. obslužná rutina) události `onclick`. Obsahuje i některé další metody, týkající se organizace tlačítek, to však pro funkčnost editoru nemá zásadní význam.

Dynamická část je nepovinná, nemusí obsahovat žádné prvky, protože prvky ve statické části jsou postačující pro plnou funkčnost aplikace. Tato část má význam pro pohodlnější operace týkajících se určitého *jmenného prostoru*. Její dynamičnost spočívá v tom, že se mění za běhu aplikace. V mém editoru má každý prostor definované své ovládací prvky, týkající se konkrétních elementů resp. atributů. Registrace těchto prvků je provedena na základě konfiguračního souboru editoru (viz 3.7.1). Ve chvíli, kdy se změní aktuální *jmenný prostor*, obnoví se dynamická část panelu tak, že obsahuje prvky zaregistrované pro nově aktivní.

3.6.3 Režim rozbalovacího stromu

Jazyk *HTML* má omezený počet komponent, pomocí kterého se dá vytvářet uživatelské rozhraní. Přestože knihovna *jQuery UI* nabízí některé další komponenty, neposkytne vývojáři zdaleka takový počet možností jako např. VCL dostupné ve vývojovém prostředí Delphi od společnosti

Borland, *toolkit* (neboli knihovna rozšiřujících funkcí) Swing dostupná v programovacím jazyku Java společnosti Sun, nebo komponenty dostupné ve známém vývojovém prostředí Visual Studio od Microsoftu. Je to zejména proto, že webové aplikace obvykle nepotřebují tak komplexní rozhraní jako aplikace desktopové.

Pod pojmem rozbalovací strom (angl. *treeview*) je myšlen prvek známý už např. z 16 bitových Windows 3.1. Jde o uživatelsky přívětivý prvek graficky zobrazující strom formou sloupce. Jednotlivé části lze pohodlně rozbalovat a zabalovat pomocí tlačítka označující příslušnou větev. Strom je v teorii grafů souvislý graf, který neobsahuje žádnou kružnici. Pomocí stromu se dá zobrazit např. souborový systém, hierarchie tříd nebo XML dokument. V JS se dají vyvíjet vlastní komponenty např. jako *pluginy* pro framework *jQuery*. Protože rozbalovací strom netvoří součást jazyka *HTML* a ani knihovny *jQuery UI*, existuje několik *pluginů* jak pro samotný JS, tak přímo pro *jQuery* realizující tuto komponentu.

Obrázek 25: Režim rozbalovacího stromu

Nakonec jsem vybral *plugin Treeview* od Jörna Zaefferera jako základ pro režim, kde se *XML* zobrazuje jako interaktivní strom. Ten je volně dostupný na [4]. Výsledek je vidět na obr. 25.

Elementy jsou ve stromě zvýrazněny tmavě modrou barvou. Atributy jsou vypsány ve stejném řádku jako element o něco světlejší barvou. Po kliknutí na atribut se zobrazí dialogové okno, kde může uživatel zadat novou hodnotu. Textové větve lze také měnit pomocí dialogového okna.

Práce s *XML* v tomto režimu je přehledná a pohodlná. Rozbalovací strom může pro jednoduché úpravy atributů a textu elementů nahradit režim zdrojového kódu, protože pomocí orientace v *XML* pomocí stromu je velmi přehledná. Tento editační mód je v editoru představován třídou `TreeView`.

3.6.4 Dialogová okna

V GUI je *dialogové okno*, neboli *dialog*, zvláštní druh okna sloužící k zobrazení zprávy, nebo k získání odezvy od uživatele. Obvykle mívají *modální* charakter, tzn. že v průběhu jejich existence blokují ostatní ovládací prvky aplikace.

V JS lze vytvářet dialogová okna pomocí nových instancí třídy `window`. Tak se vytvářejí tzv. *pop-up okna*, která prohlížeče mohou blokovat, protože mnoho webových stránek obtěžuje návštěvníky různými reklamami formou tzv. *pop-up adů* (neboli reklam zobrazených v pop-up okně). Alternativně lze pomocí DHTML (dynamického HTML) vytvářet dialogová okna bez vytváření instancí nových oken. Tento způsob využívá i knihovna *jQuery UI*, jejíž část *Dialog* se týká vytváření *dialogových oken*.

Dialog v *jQuery UI* je flexibilní nástroj s řadou možností tvořenými několika metod, vlastností a *callback* funkcemi, díky kterým se dá reagovat na různé události *dialogového okna* (např. otevření a uzavření). Díky těmto možnostem jsem realizoval dialogy GUI editoru touto formou.

3.7 Výsledná aplikace

Finální verze editoru je tvořena několika skripty, kde každý představuje konkrétní část, resp. třídu editoru. Ty je potřeba připojit k webovému dokumentu, do kterého se má editor implementovat. Skripty lze připojit v libovolném pořadí. Aby nebylo nutné připojovat několik skriptů, pokaždé když je potřeba, snažil jsem se napsat jednoduchý skript, který by jednoduše připojil všechny najednou. Bohužel prohlížeč Google Chrome zřejmě z bezpečnostních důvodů potlačuje načítání externích skriptů za běhu v JS, proto jsem od této možnosti nakonec upustil. Alternativně lze všechny skripty sloučit ručně do jednoho velkého souboru. Celý skript editoru je tvořen téměř dvěma tisíci řádky kódu.

3.7.1 Konfigurace

Abych oddělil úlohu konfigurace editoru od jeho generování, navrhl jsem pro tyto účely konfigurační *XML* skript. Můj editor má dynamický charakter, není spjatý s konkrétním jazykem a nemá pevně zadané UI. Díky tomu webový vývojář dostane flexibilní nástroj, který může přizpůsobit právě jeho potřebám. Použitím konfiguračního skriptu může pohodlně měnit chování editoru, aniž by musel zasahovat do zdrojového kódu jeho webové stránky.

V konfiguračním skriptu jsou uvedeny informace o požadované velikosti editoru a URL adresy, kde jsou informace týkající se samotného *XML* dokumentu. Jsou to: adresa *XML* dokumentu,

který je zobrazen při načtení editoru a adresa *XSL* souboru, potřebného pro převedení dokumentu na *HTML* kód zobrazitelný v prohlížeči.

Výše zmíněné informace plně postačují pro základní činnost editoru. Další možností je definice zvláštních ovládacích prvků, čili tlačítek spjatých s konkrétní funkcí. To je možné přidáním elementu `button` do konfiguračního skriptu. V něm atribut `name` určuje jméno souboru, kde je uložena ikona tlačítka, dále je zadán jako další atribut `tooltip` a poslední atribut `type` určuje typ tlačítka. Typ tlačítka `add` znamená, že tlačítko přidává do dokumentu nový element, který je definován jako obsah elementu `button`. Ten je nutné vložit do tzv. sekce *CDATA*, aby ji *parser* dál nezpracovával. Cíl kam je nový element přidán je předán v atributu `tar`. Druhou možností je hodnota `format`, která říká, že tlačítko slouží k formátování. Pokud je třeba, aby tlačítko bylo k dispozici pro určitý jmenný prostor v dokumentu, stačí definovat element `button` jako obsah elementu `namespace`. *Jmenný prostor* je zde určen pomocí atributu name, který je alias cílového *jmenného prostoru*.

```
1     <?xml version="1.0" encoding="ISO-8859-1"?>
2     <config>
3         <size>800,600</size>
4         <xmlurl>catalog.xml</xmlurl>
5         <xslurl>catalog.xsl</xslurl>
6         <schema>catalog.dtd</schema>
7         <namespace name="h">
8             <button name="para" tooltip="New Paragraph" type="add" tar="body">
9                 <![CDATA[
10                    <p>New Paragraph</p>
11                ]]>
12            </button>
13            <button name="italic" tooltip="Format Italic" type="format">
14                <![CDATA[
15                    <span style="font-style:italic">text</span>
16                ]]>
17            </button>
18        </namespace>
19        <button name="cd" tooltip="Add New CD" type="add" tar="cd">
20            <![CDATA[
21                <cd>
22                    <title>no title</title>
23                    <artist>no name</artist>
24                    <country>USA</country>
25                    <company>unknown</company>
26                    <price>0.0</price>
27                    <year>2010</year>
28                </cd>
29            ]]>
30        </button>
31    </config>
```

Obrázek 26: Konfigurační skript

Na obrázku 26 je ukázková konfigurace. Slouží pro generování editoru dříve zmíněného cd katalogu. Je zde definováno tlačítko, pomocí kterého lze přímo elementy `cd` do dokumentu

přidávat. Jmenný prostor s aliasem `h` zde představuje jazyk *XHTML*. Ten registruje element `p` (*XHTML* element odstavec) a dvě tlačítka. To první slouží k přidávání elementů `p` a pomocí druhého lze formátovat text kurzívou. Ukázka je jednoduchá a na jejím základě by se dala rozšiřovat a definovat tak nové funkce dle potřeby.

3.7.2 Testování aplikace

Během testování aplikace se ukázalo problematické optimalizovat ji pro všechny nejpoužívanější prohlížeče. Vybral jsem proto nejpopulárnější webový prohlížeč, jímž je IE od Microsoftu a zaměřil se při optimalizaci na jej. Základní komponenty editoru, mezi které patří `XmlParser`, `XsltProc` a některé další jsou plně optimalizovány. Na druhou stranu `WYSIWYG` režim je funkční pouze v IE. Zpočátku jsem se domníval, že optimalizace nebude natolik náročná část, zejména díky frameworku *jQuery*. Naneštěstí *jQuery* slučuje pouze nekompatibility prohlížečů z hlediska odlišných standardů, což pro tuto práci není postačující. V mnoha případech jsou vzájemné nekompatibility prohlížečů způsobeny jeho odlišným chování za stejné situace, odlišným zpracováním a tokem událostí a dalšími jevy, na které vývojář narazí při testování skriptu.

Uživatelské rozhraní editoru je jednoduché a přirozené. Díky použitému jazyku XML a jeho validaci má uživatel jistotu, že bude zdrojový kód dokumentu vypadat vždy jak má. Díky rozbalovacímu stromu má uživatel nad zdrojovým kódem přehled i když nezná pravidla zápisu *XML* dokumentů. *WYSIWYG HTML* editory jsou někdy kritizovány, protože zdrojový kód který vytvářejí obsahuje spoustu elementů navíc a nebo, že jejich uspořádání není vhodné. To je možné díky tomu, že v *HTML* platí poměrně volná pravidla. Naproti tomu v mém editoru si pravidla uspořádání může určit každý sám. Alternativně je možné využít již předem existující *XML* jazyk. Důležité je, aby byl tento jazyk převoditelný do *HTML* podoby. Pro ostatní jazyky by se musela použít nějaká forma vektorové grafiky pro vizualizační část editoru.

Další výhodou oproti klasickým *HTML* editorům je možnost generování různých výstupních formátů. Uživateli stačí můj *XML* editor a na straně serveru lze poměrně jednoduše na základě odeslaných *XML* provádět *XSLT* i s podporou formátovacích objektů a generovat tak formáty přímo pro tisk. Ukázka některých funkcí editoru a různých předdefinovaných *skinů* je na obr. 27.

4 Závěr

V této práci jsem ukázal, že jazyk JS je kvalitní nástroj pro vývoj rozsáhlejších projektů. Jeho přednosti je podpora OOP a dynamičnost. Při vývoji rozsáhlejších projektů je důležitá právě podpora OOP. Ta je v JS někdy kritizována, dokonce jsem narazil na názory, že OOP v JS není OOP. V kapitole 2 jsem ukázal, že JS je plnohodnotný nástroj pro OOP. Původ těchto názorů zřejmě velkou mírou pochází z faktu, že standardy pro OOP položily statické jazyky (jako je C++) a proto se může zdát některým programátorům OOP v JS neobvyklé.

Další část už byla věnována samotnému vývoji aplikace. Ukázal jsem, jakým způsobem se optimalizují skripty a co bývá častou příčinou jejich nekompatibilit. Při vývoji aplikace jsem využil volně dostupný framework pro JS *jQuery*. Ani ten však nedokáže vyřešit všechny nekompatibility, protože stejně jako JS nedokáže ovlivnit chod jádra samotného prohlížeče. Narazil jsem na situace, kdy na základě stejného vstupu produkují prohlížeče odlišný výsledek. Příkladem je práce s odstavci při WYSIWYG režimu zmíněná v kapitole 3.4.2. Během práce jsem se soustředil na to, aby byl editor plně funkční v prohlížeči IE, z důvodu jeho masové rozšířenosti. Díky modulárnímu návrhu aplikace, který jsem při jejím vývoji vytvořil, není problém orientovat se ve skriptech a dohledat nekompatibilní část a případně doplnit její implementaci tak aby řešila podporu ostatních prohlížečů.

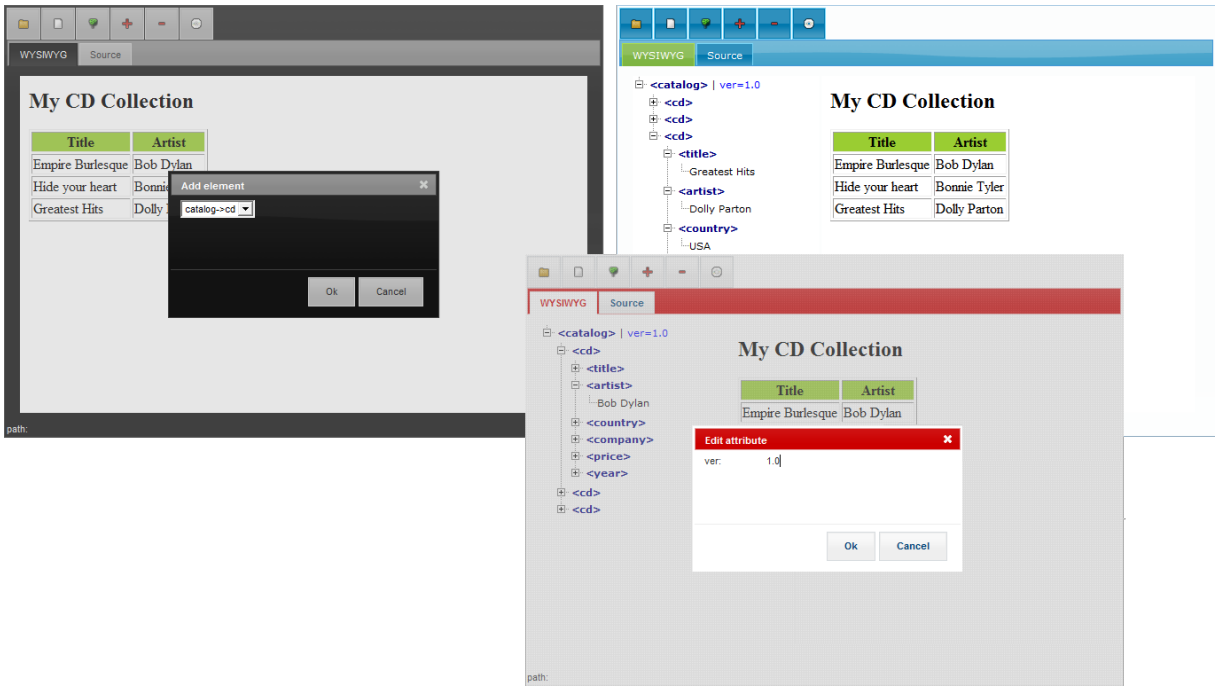
Během vývoje editoru bylo vytvořeno několik nástrojů užitečných i pro jiné aplikace. ZS se ukázal užitečným nástrojem i pro tento text, protože jsou v něm s jeho pomocí zvýrazněny ukázky kódu. Editor se dá chápat jako sada nástrojů pro práci s *XML* v jazyce JS a vývoj webových editorů, a jako nadstavbová aplikace, která tyto nástroje využívá a realizuje tak výslednou aplikaci.

Dále se v práci ukázala síla frameworku *jQuery* na poli vývoje webového uživatelského rozhraní. Možnosti které zde nabízí přibližují vývoj grafického rozhraní webových stránek postupům známým z vývoje klasických desktopových aplikací. Existuje několik konkurenčních frameworků nabízející jiné možnosti, dokonce jsou některé vyvíjeny komerčně. Díky výše zmíněným možnostem jsem se však rozhodl využít právě *jQuery*. Stručné zhodnocení aplikace uvádím v následujícím odstavci.

Výhody:

Využití XML – Jazyk *XML* nabízí oproti jazyku *HTML* více možností, např. možnost definovat vlastní jazyk, jmenné prostory a možnost provádět *XSLT* transformace.

Offline režim – Protože editor běží na straně serveru, je možné jej plně využívat i bez přístupu k internetu.



Obrázek 27: Ukázka aplikace

Dynamické GUI – Webový vývojáři mohou komunikovat s návštěvníky jejich stránek a nabídnout jim takové ovládací prvky, které požadují, aniž by jakkoliv zasahovaly do zdrojových kódů aplikace formou vlastních ovládacích prvků. Ty se aktualizují za běhu na základě právě aktivního jmenného prostoru.

Režim rozbalovacího stromu – Užitečný ovládací prvek, který není obvykle přítomen u webových editorů.

Zvýrazňovač syntaxe – Snadno konfigurovatelný zvýrazňovač syntaxe, který lze využít samostatně pro generování statického obsahu, nebo zabudovat do jiných editorů.

Nevýhody:

Optimalizace – Editor je bohužel plně funkční pouze v prohlížeči IE od společnosti Microsoft, u zvýrazňovače nefunguje zcela optimálně nastavování pozice kursoru (platí pouze pro editační režim).

Pouze DTD – Na straně klienta a tím pádem i v mém editoru lze validovat pouze DTD schémata. Na druhou stranu je editor napsán tak, že lze poměrně jednoduše připojit externí validátor (např. běžící na straně serveru), který by problém řešil.

Velikost – Zdrojové kódy editoru jsou na poměry JavaScriptu poměrně rozsáhlé, dá se částečně řešit automatizovanými nástroji pro minimalizaci JavaScriptu.

V práci by se dalo dále pokračovat, existuje řada funkcí nebo možností, které by mohl editor implementovat.

Literatura

[1] FLANAGAN, David. *JavaScript : The Definitive Guide*. 4th Edition. USA : O'Reilly Media, 2001. 461 s. ISBN 978-0-596-00048-6.

[2] *W3schools* [online]. 1999 [cit. 2010-05-11]. XSLT Tutorial. Dostupné z WWW: <http://www.w3schools.com/xsl/>.

[3] *JQuery UI* [online]. 2006 [cit. 2010-05-18]. ThemeRoller. Dostupné z WWW: <http://jqueryui.com/themeroller/>.

[4] ZAEFFERER, Jörn. *Bassistance* [online]. 2009 [cit. 2010-05-18]. JQuery: Treeview. Dostupné z WWW: <http://bassistance.de/jquery-plugins/jquery-plugin-treeview/eroller/>.

[5] WILTON-JONES, Mark. *How To Create* [online]. 2001 [cit. 2010-05-21]. JavaScript Tutorial. Dostupné z WWW: <http://www.howtocreate.co.uk/tutorials/javascript/important>.

[6] STEIGERWALD, Daniel. *Zdroják.cz* [online]. 2010 [cit. 2010-05-21]. Seriál OOP v Javascriptu. Dostupné z WWW: <http://zdrojak.root.cz/serialy/oop-v-javascriptu/>.

Obsah CD

/dp/LaTeX/ — adresář s diplomovou prací v latexu

/dp/pdf/ — adresář s diplomovou prací ve formátu PDF

/editor/ — adresář se zdrojovými kódy editoru

/editor/index.html — jednoduchá ukázka editoru

/zs/index.html — ukázka zvýrazňovače syntaxe

/cti.txt — popis obsahu CD a pokyny pro instalaci editoru